DOS EXTRA Nr. 8 – 1989

DOS EXTRA

18,– DM 145 öS 18,– sfr 26,50 Fm

Sonderausgabe DOS EXTRA

Nr. 8 1989

ISSN 0934-2842

DMV-Verlag

Eine Publikation der DOS INTERNATIONAL

Sensationell

DOS-Assembler SAM III

Durchblick total

- TSR-Programmierung
- Arithmetikprozessor
- V20/V30-Prozessor

Mit Assembler im Duett

- Basic
- Pascal
- dBase
- C

Werkstatt

- AFD-Pro
- MASM
- TASM

Assembler auf einen Blick

- Grundlagen
- Tabellen
- Referenzen

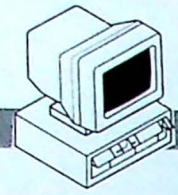
Assembler-Programmierung von Anfang an

CPS

CPS Computertechnik GmbH
Hamburger Str. 283 • 3300 Braunschweig
Fax (05 31) 33 06 61 • BTX *20088 1490#

Ladenöffnungszeiten: Mo-Fr 9-13 und 14-18 Uhr
Sa 9-14 Uhr
Studentenrabatte geg. Nachweis a. Anfrage !

Tel. (05 31) 33 06 63



CPS XT und AT

CPS XT I

10 MHz, 256 KB RAM bis 1 MB on board. Sockel für Co-Prozessor, Dualgrafikk. (Herc. & Color), Multi I/0, 1 Drive 5,25" 360k, 20 MB Festplatte ser./par., Game, Clock, 14" Flat Screen Monitor s/w, ... 2000,-

CPS AT I (siehe Test DOS 1/89)

80286-12 CPU 8/10/12 MHz ,
640 KB RAM bis 4MB on board. 0-Wait, Sockel für Co-Prozessor, EMS 4.0, Phoenix Bios 64 K ROM mit Setup-Funktion, Uhr u. Kalender, 1 ser./1 par./Game-Port, Dualgrafikkarte (Herc. & Color), 20 MB Festpl. 38ms, 1 Drive 5,25" 1,2 MB und 1 Drive 3,5" 720K/1,44MB, Cherry Tastatur mit sep. Cursorblock, 14" Flat Screen Monitor s/w oder amber, MS-DOS 3.3 und GW-Basic 3540,-

CPS AT Suntac (siehe Testbericht MegaByte 1/89)

wie AT I jedoch Festplatte 28ms, Slimline Gehäuse, ... 3740,-

CPS AT I Mini-Tower

80286-12 CPU 8/12 MHz, 640 KB RAM bis 4 MB on board. EMS 4.0, 0-Wait, AMI Bios, Page/Interleave Speicher Controll, sonst wie AT I 3850,-

CPS AT II

80286-16 CPU 8/16 MHz,
640 KB RAM bis 4 MB on board,
EMS 4.0, 0-Wait, sonst wie AT I 4050,-

CPS AT II Tower (siehe Testbericht FORMAT 5/89)

80286-16 CPU 8/10/12/16 MHz,
1 MB RAM bis 4 MB on board, 40 MB Festpl. 40ms, EMS, 0-Wait, Sockel für 80287, Phoenix Bios 64 K ROM mit Setup-Funktion, Uhr u. Kalender, 1 ser./1 par./Port, Herc. komp. Grafikk., 1 Drive 5,25" 1,2 MB und 1 Drive 3,5" 720 K/1,44 MB, Cherry Tastatur m. sep. Cursorbl., 14" Flat Screen Monitor, Mouse RS 232, MS-DOS 3.3 und GW-Basic, Gem 3.01 5740,-

CPS AT III

80386-20 CPU 20 MHz,
2 MB RAM bis 16 MB on board. 0-Wait, Sockel für Co-Prozessor. Phoenix Bios 64 K ROM, DUAL-Grafikk. (Herc. & Color), Uhr u. Kalender, 1 ser./1 par./Maus Port, 1 Drive 5,25" 1,2 MB und 1 Drive 3,5" 720K/1,44 MB, 20 MB Festplatte 28ms, 102 MF II Cherry Tastatur, MS-DOS 3.3 und GW-Basic, GEM 3.01, Q-Mouse, 14" Flat Screen Monitor s/w, ... 7470,-

Gerätekonfigurationen nach Ihren Wünschen veränderbar. Speichererweiterungen bitte Tagespreis anfragen.

18 Monate Garantie auf CPS Rechner!

Disketten

NN 2DD 3,5" 10 Stck.	20,-
NN 2DD 5,25" 10 Stck.	9,-
Magix MF 2DD 3,5"	24,50
Select MF 2DD 3,5"	33,50
Select MD 2DD 5 1/4"	19,-
Fuji MD 2DD 5 1/4"	27,-

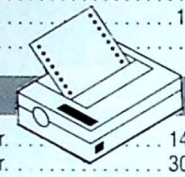
FARBBÄNDER

Kores

Mindestabnahme 3 Stück

STAR NL/NG/ND/NR-10, Stück	12,-
EPSON LX-800/LQ-500, Stück	11,-
PANASONIC KX-P, Stück	13,-
OKI ML 320	14,50
OKI ML 390	16,50
NEC 2200	13,50
NEC P6+/P7+	15,30
Star LC 10	12,-

DRUCKER



OKI Microline 390 Centr.	1448,-
OKI Microline 393 Centr.	3079,-
OKI Microline 320	1285,-
OKI Microline 321	1798,-
OKI Microline 391	2498,-
NEC P 6 Plus Centr.	1798,-
NEC P 7 Plus Centr.	1985,-
NEC P 2200 Centr.	898,-
STAR LC 10 Centr.	599,-
EPSON LX 800 Centr.	**549,-**
EPSON LQ 500 Centr.	992,-
EPSON LQ 1050 Centr.	1950,-
Mannesmann Tally, mit Einzelbl.einz.	699,-
Sheet Feeder	
OKI 320/390	445,-
NEC P6+	541,-
NEC P7+	705,-
EPSON LX 800	225,-
EPSON LQ 500	225,-
Centronics Druckerkabel	**18,80**

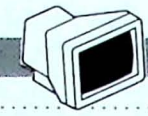
Weitere Drucker auf Anfrage

Wir liefern **nur** mit dt. Handbuch, Seriennummer und Herstellergarantie !!!Drucker-Grauimporte mit engl. Handbuch, ohne Seriennummer, ohne Herstellergarantie sind bei uns ausgeschlossen.

C= COMMODORE

AMIGA 500	930,-
AMIGA 2000	1830,-
AMIGA 500 + Mon. 1084 S	1510,-
AMIGA 2000 + 20 MB Harddisk SCSI.	2640,-
LW ext. AMIGA 500 3,5"	290,-
LW ext. AMIGA 2000 5,25"	419,-
LW int. AMIGA 2000 3,5"	220,-
RGB Commodore 1084 S	630,-
RGB Stereo Philips CM 8833	630,-
PC 10/III, 20 MB Festpl. MS-DOS	2200,-
PC 20/III, 20 MB Festpl., MS-DOS	2300,-
PC/AT-Karte	2400,-
PC/XT-Karte	880,-
A 2058 2 MB RAM-Erweiterung	1180,-
A 501 512 MB RAM-Erweiterung	330,-

MONITORE



EGA Standard	815,-
EIZO 9060s	1855,-
NEC MULTISYNC II	1500,-
NEC MULTISYNC GS	630,-
TTL 14" Flat Screen m. Fuß sw/amber	225,-
TTL 12" Flat Screen m. Fuß sw/amber	180,-
Mitsubishi 1481	1398,-

weitere Monitore auf Anfrage

Copam 80286-C-120

80286 12 MHz,
0-Wait, 640 KB RAM,
Sockel 80287, Hercules komp. MGA/CGA Grafikkarte, Q-Mouse, Uhr u. Kalender, 2 ser./2 par. Port, Phoenix Bios 64 K ROM mit Setup-Funktion, 1 Drive 1,2 MB 5,25" 20 MB Festplatte, MF-II Tastatur mit 102 Tasten, Formschönes Gehäuse mit Betriebs-LED's, Turbo, Reset und Schlüsselschalter, 14" Flat Screen Monitor s/w mit Dreh- und Schwenkfuß, Software, MS-DOS 3.3 und GW-Basic Handbuch, GEM 3.01 Benutzeroberfläche + Textverarbeitung 1st Word Plus
Garantie 12 Monate
Komplett-Preis 3480,-

Festplatten und Laufwerke

SEAGATE ST 125-0, 20 MB, 38ms	549,-
SEAGATE ST 125-1, 20 MB, 28ms	558,-
SEAGATE ST 251-0, 40 MB, 40ms	924,-
SEAGATE ST 251-1, 40 MB, 28ms	999,-
Controller NCL 5525 HDD/FDD	290,-
Laufwerk 360 KB 5 1/4" TEAC	190,-
Laufwerk 1,2 MB 5 1/4" TEAC	215,-
Laufwerk 1,44 MB 3,5" TEAC	235,-

Sämtliche Angebote freibleibend, Zwischenverkauf vorbehalten. Wir liefern an Nicht-Kaufleute nur per UPS-Nachnahme mit einem Rückgaberecht innerhalb von 10 Tagen (nur Hardware). Ins Ausland nur per Vorkasse. Fordern Sie unseren Gesamtkatalog gegen 3,-DM in Briefmarken. Es gelten unsere allgemeinen Geschäftsbedingungen.

Wir akzeptieren:

AMERICAN EXPRESS · VISA · Diners Club

Nutzen Sie unseren Bequem-Kauf-Kredit!

Wir sind autorisierter Fachhändler für:

ARCHE TECHNOLOGIES · Acer · Copam Electronics Corp. · Commodore · EPSON · OKI COMPUTERDRUCKER

Geschwindigkeit ist keine Hexerei

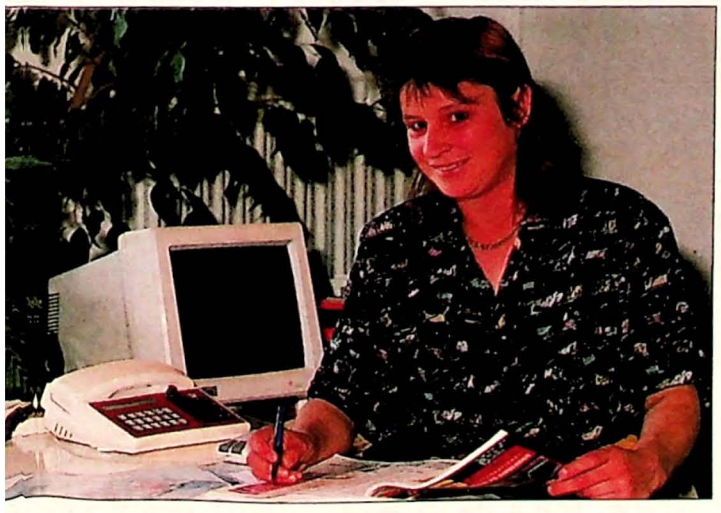
Im Zeitalter der Hochsprachen, in dem in Sprachen wie C, Turbo Pascal oder gar Prolog programmiert wird, sieht der eine oder andere aus verständlichen Gründen vielleicht nicht ein, warum er denn noch Assembler lernen soll. Doch näher besehen, gibt es einige einleuchtende Gründe dafür.

Ein Argument ist wohl die Faszination, direkt mit der Hardware, dem Prozessor des Computers, zu kommunizieren. Damit bietet Ihnen die Assemblersprache Zugriff auf das gesamte Leistungsspektrum Ihres Computers. Die grundlegenden Kenntnisse der Hardware, die Assemblerprogrammierung voraussetzt, führen zudem Hand in Hand zu einem besseren Verständnis der Hochsprachen wie Basic oder C.

Den Ausschlag, sich für Assembler zu entscheiden, gibt für viele Programmierer sicherlich der Zeitfaktor. Mögen auch Hochsprachen den größeren Programmierkomfort bieten, so läuft ihnen, sind schnelle Routinen gefragt, die Assemblersprache fraglos den Rang ab. Hinzu kommt, daß Sie in keiner Sprache so kurze Programmcodes erzeugen wie in Assembler! In einer Zeit, da Programme immer komplexer werden, sicher ein Argument, das überzeugt.

Zugegeben, in Assembler zu programmieren ist kein Kinderspiel, setzt es doch profunde Kenntnisse der Computer-Hardware voraus. Zudem sind kompakte Informationen zum Thema Maschinensprache selten, meist muß man sie sich mühsam aus zahlreichen verschiedenen Quellen zusammensuchen.

Aus der Intention, diesen Mißstand zu beheben, und aus der Tatsache, daß sich erstaunlich viele unserer Leser mit der Materie Assembler auseinandersetzen und Interesse daran bekunden, wurde der Gedanke geboren, dieses Sonderheft dem Thema Assembler zu widmen. In der vorliegenden Ausgabe finden Sie Informationen zum Programmieren in Assembler in übersichtlicher und komprimierter Form.



Den Kern bildet ein ausführlicher Referenzteil, der tabellarisch Prozessorbefehle, Registersatz, Adressierungsarten, das BIOS-Datensegment und andere für die Assemblerprogrammierung unentbehrliche Daten aufführt.

Großen Wert gelegt haben wir auch auf eine Einführung in Assembler für den Neuling. Neben ersten Einblicken wird dem Einsteiger Basiswissen über Grafik- und Soundprogrammierung, Arithmetik, Interrupts, Funktionen sowie Pseudobefehle leicht verständlich vermittelt.

Weitere Grundlagenthemen, wie TSR- und 8087-Programmierung, Speicherverwaltung und ähnliches mehr, sind ausführlich dargestellt.

Wer nicht ausschließlich in Assembler programmiert, dem kommt es in erster Linie darauf an, seine C- oder Pascal-Programme mit schnellen Assemblerroutinen zu optimieren. Auch dieser Problematik wurde Rechnung getragen: Alle C-, Basic-, Pascal- und selbst dBase-Programmautoren erhalten Informationen über die effektive Einbindung von Assembler in »ihre« Sprache.

Ein diffiziles Thema ist uns immer eine Werkstatt wert, erst recht in DOS Extra: Debugger, MASM, TASM und AFD lauten die Themen. Wer fertige Programme dem »Selberbasteln« vorzieht, wird ebenfalls bedient: Ein Update des »DOS-eigenen« Assemblers SAM, die Version SAM III, stellt sich vor.

Wir hoffen, mit der DOS Extra 8 allen Lesern, die sich mit dem faszinierenden Sujet Assembler bereits auseinandersetzen oder auseinandersetzen wollen, ein wertvolles Hilfsmittel an die Hand zu geben.

Ihre

Eva Hierlmeier

Inhalt

Eingabehilfe

SCheck 2.01:
Checksummer für sichere Listing-Eingabe 6
– *Eingabehilfe*

Einstieg in Assembler

Assembler zum ersten... 10
– *Einführung*

Dateitypen und MS-DOS-Funktionen 16

Die Pseudobefehle des MASM 4.0 und SAM III 24

Das Einmaleins des Prozessors 28
– *Arithmetik*

Assembler am Zeichenbrett 32
– *Grafik*

Musikalische Maschinensprache 38
– *Sound*

Grundlagen

Speicherverwaltung unter MS-DOS 40

Optimale TSR-Programmierung in Assembler 43

Des Assemblers neue Kleider 48
– *Segmentbefehle des MASM 5.x*

Arithmetikprozessoren 54
– *Heinzelmännchen der CPU*

Die (un)heimlichen Helfer von MS-DOS 57
– *Gerätetreiber*

Der geklonte Prozessor 62
– *V20/V30*

Referenz

Die Assemblerbefehle des 8086/8088 64

Die zweite Stufe: der 80286 72

Die dritte Stufe: der 80386 75

8087 – das ungeliebte Genie 78

Der Registersatz des 8086/80286-Prozessors 82

Die Speicheradressierung des 8086/8088 84

Assemblerreferenz: Programmablaufsteuerung 88

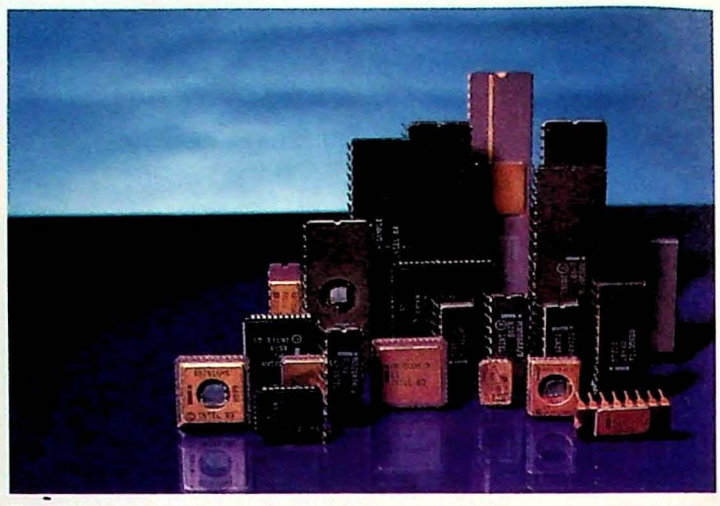
Assemblerreferenz: Speicher-, Port- und Stapelzugriff 90

Assemblerreferenz: Zeichenketten 92

Assemblerreferenz: Bitmanipulation und Logik 98

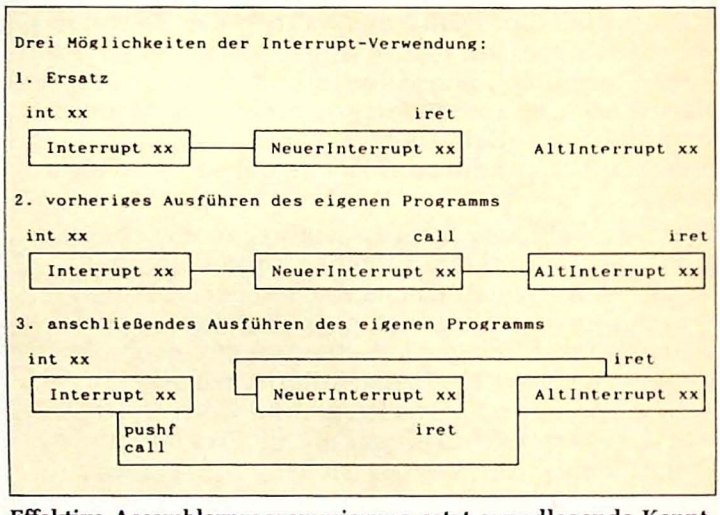
Assemblerreferenz: Arithmetik 102

Top Secret: das BIOS-Datensegment 106



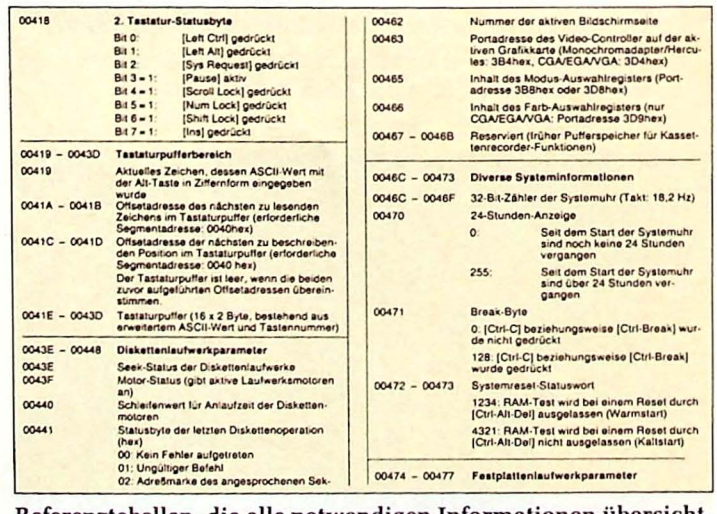
Den Einstieg in die Assemblerprogrammierung erleichtern ausführliche Informationen über Grafik, Arithmetik, Interrupts und vieles mehr

Seite 10



Effektive Assemblerprogrammierung setzt grundlegende Kenntnisse voraus. Alles über TSR, Speicherverwaltung und und und erwartet Sie ab

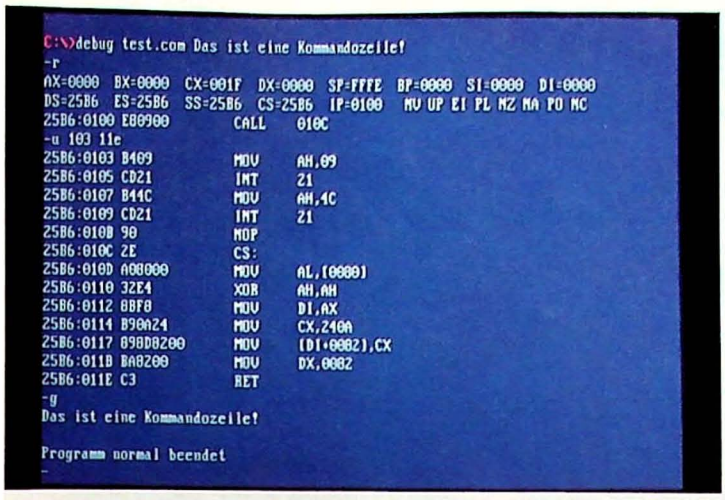
Seite 40



00418	**2. Tastatur-Statusbyte**	
	Bit 0:	[Left Ctrl] gedrückt
	Bit 1:	[Left Alt] gedrückt
	Bit 2:	[Sys Request] gedrückt
	Bit 3 = 1:	[Pause] aktiv
	Bit 4 = 1:	[Scroll Lock] gedrückt
	Bit 5 = 1:	[Num Lock] gedrückt
	Bit 6 = 1:	[Shift Lock] gedrückt
	Bit 7 = 1:	[Ins] gedrückt
00419 – 0043D	**Tastaturpufferbereich**	
00419	Aktuelles Zeichen, dessen ASCII-Wert mit der Alt-Taste in Ziffernform eingegeben wurde	
0041A – 0041B	Offsetadresse des nächsten zu lesenden Zeichens im Tastaturpuffer (erforderliche Segmentadresse: 0040hex)	
0041C – 0041D	Offsetadresse der nächsten zu beschreibenden Position im Tastaturpuffer (erforderliche Segmentadresse: 0040 hex) Der Tastaturpuffer ist leer, wenn die beiden zuvor aufgeführten Offsetadressen übereinstimmen	
0041E – 0043D	Tastaturpuffer (16 x 2 Byte, bestehend aus erweitertem ASCII-Wert und Tastennummer)	
0043E – 0044B	**Diskettenlaufwerkparameter**	
0043E	Seek-Status der Diskettenlaufwerke	
0043F	Motor-Status (gibt aktive Laufwerksmotoren an)	
00440	Schleifenwert für Anlaufzeit der Diskettenmotoren	
00441	Statusbyte der letzten Diskettenoperation (hex) 00: Kein Fehler aufgetreten 01: Ungültiger Befehl 02: Adreßmarke des angesprochenen Sek-	
00462	Nummer der aktiven Bildschirmseite	
00463	Portadresse des Video-Controller auf der aktiven Grafikkarte (Monochromadapter/Hercules: 3B4hex, CGA/EGA/VGA: 3D4hex)	
00465	Inhalt des Modus-Auswahlregisters (Portadresse 3B8hex oder 3D8hex)	
00466	Inhalt des Farb-Auswahlregisters (nur CGA/EGA/VGA; Portadresse 3D9hex)	
00467 – 0046B	Reserviert (früher Pufferspeicher für Kassettenrecorder-Funktionen)	
0046C – 00473	**Diverse Systeminformationen**	
0046C – 0046F	32-Bit-Zähler der Systemuhr (Takt: 18,2 Hz)	
00470	24-Stunden-Anzeige	
	0:	Seit dem Start der Systemuhr sind noch keine 24 Stunden vergangen
	255:	Seit dem Start der Systemuhr sind über 24 Stunden vergangen
00471	Break-Byte	
	0: [Ctrl-C] beziehungsweise [Ctrl-Break] wurde nicht gedrückt	
	128: [Ctrl-C] beziehungsweise [Ctrl-Break] wurde gedrückt	
00472 – 00473	Systemreset-Statuswort	
	1234: RAM-Test wird bei einem Reset durch [Ctrl-Alt-Del] ausgelassen (Warmstart)	
	4321: RAM-Test wird bei einem Reset durch [Ctrl-Alt-Del] nicht ausgelassen (Kaltstart)	
00474 – 00477	**Festplattenlaufwerkparameter**	

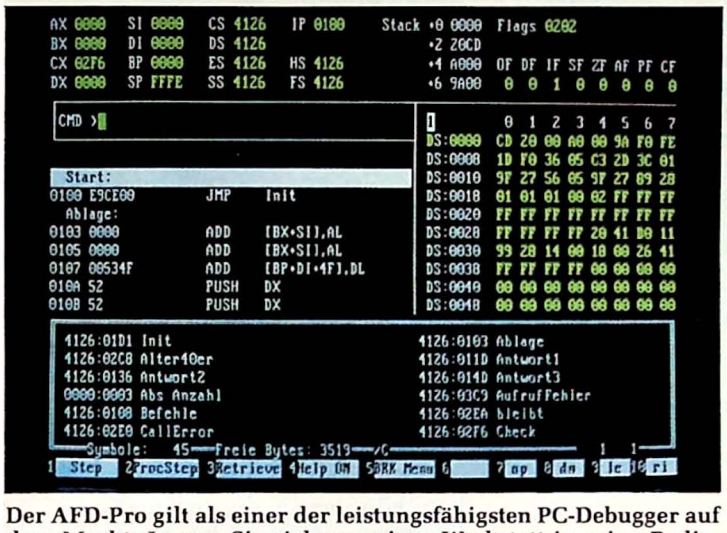
Referenztabellen, die alle notwendigen Informationen übersichtlich auflisten, gehören zum täglichen Brot des Assemblerprogrammierers

Seite 64



Der MS-DOS-Debugger ist ein wichtiges Werkzeug für Assemblerprogrammierer. Eine Werkstatt vermittelt informative Tips

Seite 134



Der AFD-Pro gilt als einer der leistungsfähigsten PC-Debugger auf dem Markt. Lassen Sie sich von einer Werkstatt in seine Bedienung einführen

Seite 142

SAM III ist da, die neueste Version des »DOS-eigenen« Makro-Assemblers zum Abtippen: Assemblerprogrammierung zum Nulltarif!

Seite 160

Hochsprachen und Assembler

Assemblerprogramme in GW-Basic **110**
– *ein effektives Gespann*

Maschinen-Basic **114**
– *Assembler in Quick Basic einbinden*

Tuning für Turbo **117**
– *Einbinden von Maschinenprogrammen in Turbo Basic*

Turbo Pascal mit Assembler **120**
– *die perfekte Ergänzung*

Assembler und C als Duo Infernale **126**

dBase-Programme »tunen« mit Assemblermodulen **130**

Werkstatt

Auf Fehlersuche mit dem Debugger **134**

Die MASM-5.0-Werkstatt **136**

Die AFD-Pro-Werkstatt **142**

Die Turbo-Assembler-Werkstatt **150**

Utilities

Von »exe« nach »com« **156**
– *Ersatz für »Exe2bin«*

SAM III

Makroassembler SAM III **160**
– *Der DOS-Assembler zum Nulltarif*

Rubriken

Geschwindigkeit ist keine Hexerei **3**
– *Editorial*

Bücher zur Assemblerprogrammierung **109**

Zum Thema Listing **149**
– *ohne Spielregeln geht es nicht*

Impressum **178**

Inserenten **178**

SCheck 2.01: Checksummer für sichere Listing-Eingabe

Die Programme in der DOS International und der Sonderheftreihe DOS Extra werden nach einem speziellen Verfahren mit Prüfsummen versehen und abgedruckt. Das für die Überprüfung notwendige Prüfsummenprogramm »SCheck« liegt jetzt auch als Version 2.01 in Basic vor.

Alle in der DOS Extra und der DOS International abgedruckten Programme sind mit Prüfsummen und Zeilennummern versehen. Jede Programmzeile beginnt mit einer dreistelligen Prüfsumme. Sie ist in eckige Klammern gesetzt und wird beim Abtippen nicht mit eingegeben. Nach der Klammer folgt ein Leerzeichen, und darauf der eigentliche Text der Programmzeile (nur bei Basic-Programmen) oder eine Zeilennummer, die Sie ebenfalls nicht mit eingeben, und darauf erst der Programmtext.
Nur diesen letzten Teil der abgedruckten Zeile geben Sie beim Abtippen ein.
Bei allen Programmzeilen wird im Abdruck nach jeweils 55 Zeichen ein Zeilenumbruch vorgenommen. Die umgebrochene Teilzeile erhält dabei keine Prüfsumme. Bei der Eingabe geben Sie umgebrochene Zeilen in einem Stück ein, drücken also nicht die Enter-Taste wenn ein Zeilenumbruch im Ausdruck steht, sondern nur, wenn eine neue Zeilennummer auftaucht.
Beim Abtippen von Programmen sind Tippfehler kaum zu vermeiden. Mit dem Checksummenprogramm »SCheck« überprüfen Sie die abgetippten Listings auf eventuelle Fehler. Das Programm erlaubt nicht nur die Diagnose von fehlerhaft eingegebenen Zeilen, sondern mittels eines integrierten Editors auch gleich die Korrektur der fehlerhaften Zeilen.
SCheck wird unter GW-Basic mit

```
run "scheck"
```

gestartet. Als erstes wählen Sie die gewünschte Prüfsummenversion. In der vorliegenden DOS-Ausgabe brauchen Sie immer »SCheck Version 2.0«. Nur wer Programme aus Ausgaben der DOS International vor Oktober 1988 eingeben will, muß den ebenfalls integrierten Prüfsummenalgorithmus »SCheck Version 1.0« aufrufen. Danach geben Sie den gewünschten Prüfmodus – normalerweise »Standard« – an und abschließend den Programmnamen des Listings, so wie Sie es auf dem Datenträger abgelegt haben.
Zuvor sollten Sie sich aber sicherheitshalber eine Kopie Ihres Listings anfertigen. SCheck arbeitet zwar nicht mit dem Original, sondern mit einer temporären Zwischendatei und verändert die Originaldatei erst ganz zum Schluß, nach dem erfolgreichen Abschluß der Arbeit. Dennoch sollten Sie auch ein noch so geringes Risiko eines Datenverlusts, etwa durch Fehler beim Speichern, ausschalten.
Wie Sie im Bild sehen, zeigt SCheck die aktuelle Zeile stets in einem besonderen Bildschirmfenster an. Die Prüfsumme erscheint links unter dem Fenster. Beim Überprüfen Ihres Listings vergleichen Sie die angezeigte Prüfsumme mit der in der DOS-Ausgabe in eckigen Klammern abgedruckten Prüfsumme. Rechts daneben finden Sie die aktuelle Zeilennummer und den Dateinamen. Im unteren Fenster werden Fehlermeldungen und Hinweise auf die weitere Bearbeitung ausgegeben. Im abgebildeten Fall können Sie mit [Enter] die Zeile übernehmen oder mit [E] den Editor aufrufen.
Der Editor dient der Korrektur fehlerhafter Zeilen. Er arbeitet immer im Einfügemodus, so daß fehlerhafte Eingaben extra gelöscht werden müssen. Sie können also nichts versehentlich überschreiben. Mit den Cursor-Tasten steuern Sie den Cursor im Editierfenster. [Home] bringt den Cursor an den Anfang der angezeigten Zeile, [End] entsprechend ans Ende.
Löschen können Sie das Zeichen links vom Cursor mit der großen Löschtaste oberhalb der Enter-Taste, [Del] entfernt das Zeichen unter dem Cursor. Mit [Enter] übernehmen Sie die korrigierte Zeile. Steuerzeichen wie zum Beispiel Tabulatoren dürfen in einer Zeile nicht verwendet werden. Haben Sie so das gesamte Programm oder auch nur »verdächtige« Teile davon durchgearbeitet, so speichert ein letzter Druck auf die Enter-Taste das korrigierte Listing. SCheck wird beendet und das fehlerfreie Programmlisting steht zu Ihrer Verfügung.
SCheck erlaubt aber noch einen zweiten Weg, das Listing korrekt einzutippen. Dabei geben Sie die Prüfsumme gleich bei der Eingabe ein. Die eckigen Klammern lassen Sie aber weg. Hinter der dreistelligen Prüfsumme fügen Sie ein Leerzeichen ein und danach den normalen Programmtext dieser Zeile. Wenn Sie den Text so aufbereitet vorliegen haben, können Sie die automatische Überprüfung mit SCheck nutzen. Rufen Sie dazu die Option »Mit Prüfsumme« auf.
SCheck verwandelt das Listing nun in eine Quelldatei für den jeweiligen Compiler oder Interpreter, indem die drei Prüfsummenstellen und das Leerzeichen vom Rest der Zeile abge-

Von 2.0b zu 2.01

Als aufmerksamer Leser haben Sie sicher bereits bemerkt, daß die Nummer der hier vorgestellten SCheck-Version in Gegensatz zu der Version im letzten DOS Extra 2.01 lautet. Dieser »Update« ist nötig geworden, weil die Version 2.0b bei der Bearbeitung von sehr langen Programmzeilen einen »Overflow Error« erzeugt.
Falls Sie die Version 2.0b aus der letzten DOS Extra abgetippt haben, müssen Sie nur zwei kleine Änderungen vornehmen, um zur Version 2.01 zu gelangen. Ersetzen Sie dazu in Programmzeile 10070 den Ausdruck

```
DEFINT A-Z
```

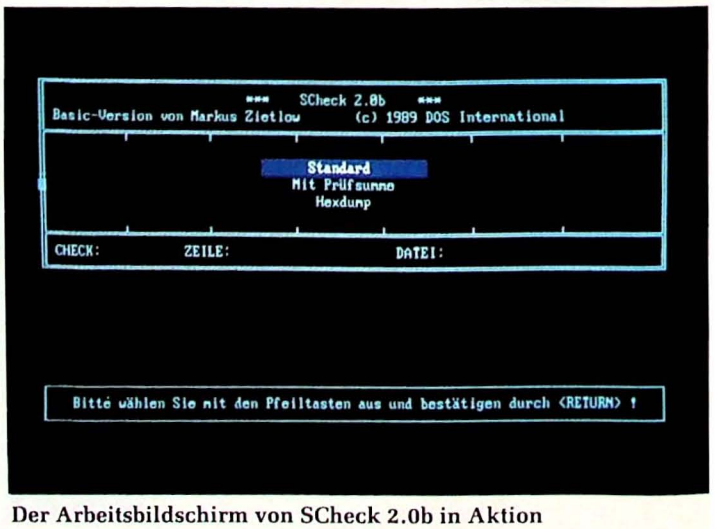
durch

```
DEFINT A-O,Q-Z:DEFSNG P
```

und fügen Sie die Zeile

```
12945 IF P>32767 THEN P = P - 32768
```

neu in das Programm ein.



Der Arbeitsbildschirm von SCheck 2.0b in Aktion

trennt werden. Aus der so entstandenen Programmzeile berechnet SCheck die Prüfsumme und vergleicht sie mit der zuvor abgetrennten Summe, die Sie eingegeben haben. Falls ein Unterschied auftritt, haben Sie Gelegenheit, die fehlerhafte Zeile manuell zu korrigieren.

Nach einem solchen Durchlauf von SCheck steht wieder das fehlerfreie Programm, diesmal allerdings ohne Prüfsumme, auf der Diskette. Auch hier gilt, daß Sie nicht mit dem Original arbeiten sollten, sondern nur mit einer Kopie, um im Zweifelsfall nochmal von vorne beginnen zu können.

SCheck verfügt aber noch über zwei weitere Optionen. Die von Fall zu Fall abgedruckten hexadezimalen Listings können Sie nämlich mit SCheck im Binärmodus direkt in lauffähige Programme umwandeln. Dazu müssen Sie das Hexdump-Listing mit einem ganz normalen Editor (zum Beispiel Turbo Pascal, Wordstar oder Edlin) als Textdatei eingeben. Anschließend überprüfen Sie mit SCheck zunächst die Korrektheit der abgetippten Daten wie bei jedem anderen Listing. Befindet sich das Programm korrekt auf Diskette, so wird die Option »Hexdump« aufgerufen. SCheck wird nun dazu veranlaßt, eine Hexdump-Textdatei in ein ablauffähiges Programm umzuwandeln. SCheck erzeugt in jedem Fall eine Datei mit der Namenserweiterung »bin«. Um daraus eine lauffähige Datei zu machen, müssen Sie sie mit dem Ren-Befehl von MS-DOS umbenennen:

```
ren <name>.bin <name>.com
```

Falls eine Exe-Datei erzeugt werden soll (bitte entnehmen Sie dies der jeweiligen Programmbeschreibung) lautet der entsprechende Befehl:

```
ren <name>.bin
<name>.exe
```

Sie sehen, mit SCheck 2.0b ist das korrekte Abtippen von Programmlistings kein bißchen komplizierter geworden – aber durch einen verbesserten Prüfsummen-Algorithmus um einiges sicherer.

(ev/ma)

SCheck mit SBasic compilieren

Um die Eingabehilfe SCheck mit dem SBasic-Compiler compilieren zu können, müssen Sie lediglich aus Programmzeile 10070 die Zeichenfolge »key off: « löschen und die Zeilen 10720, 11240 sowie 11110 bis 11170 vollständig entfernen, so daß damit die nicht unbedingt notwendige Fehlerbehandlung durch den On-Error-Befehl, den SBasic nicht verarbeiten kann, entfällt.

Name	SCheck.bas
Sprache	GW-Basic, SBasic
Grafikkarte	jede
Besonderheiten	keine

```
[SS5] 1ØØØØ '****************************************
      *******
[F29] 1ØØ1Ø '* SCheck - Prüfsummengenerator für DOS-Lis
      tings *
[ML4] 1ØØ2Ø '*
      *
[8Ø7] 1ØØ3Ø '*            Konzeption: Volker Everts
      *
[AC7] 1ØØ4Ø '*          Basic-Version: Markus Zietlow
      *
[XS5] 1ØØ5Ø '****************************************
      *******
[4UØ] 1ØØ6Ø '
[FL7] 1ØØ7Ø KEY OFF: DEFINT A-O,Q-Z: DEFSNG P: COLOR 7,
      Ø,1
[5G8] 1ØØ8Ø T = 1: TEMP = 2: F = 3'Ein-/Ausgabekanäle f
      estlegen
[332] 1ØØ9Ø FALSE = Ø
[9Ø2] 1Ø1ØØ TRUE = -1
[EY7] 1Ø11Ø ZEICHEN$ = "Ø123456789ABCDEFGHKLMNPRSTUVWXY
      Z"
[9H2] 1Ø12Ø LAUF = FALSE
[6B2] 1Ø13Ø BIN = FALSE
[851] 1Ø14Ø CLS
[9S1] 1Ø15Ø PRINT
[KØ7] 1Ø16Ø PRINT CHR$(2Ø1); STRING$(78, 2Ø5); CHR$(187
      );
[9FC] 1Ø17Ø PRINT CHR$(186); STRING$(26, 32); "***    S
      Check 2.Ø1    ***"; STRING$(27, 32); CHR$(186);
[Z7G] 1Ø18Ø PRINT CHR$(186); " Basic-Version von Markus
       Zietlow       (c) 1989 DOS International
          "; CHR$(186);
[C5C] 1Ø19Ø PRINT CHR$(2Ø4); "═══════════╧═══════════
      ═══════════╧═══════════╧═══════════╧═══════════
      ═══"; CHR$(185);
[DR3] 1Ø2ØØ FOR ZAEHLER = 1 TO 5
[ZY6] 1Ø21Ø   PRINT CHR$(186); STRING$(78, 32); CHR$(18
      6);
[CR2] 1Ø22Ø NEXT ZAEHLER
[84C] 1Ø23Ø PRINT CHR$(2Ø4); "═══════════╧═══════════
      ═══════════╧═══════════╧═══════════╧═══════════
      ═══"; CHR$(185);
[ØØD] 1Ø24Ø PRINT CHR$(186); " CHECK:           ZEILE:
                          DATEI:
          "; CHR$(186);
[PØ7] 1Ø25Ø PRINT CHR$(2ØØ); STRING$(78, 2Ø5); CHR$(188
      );
[2D2] 1Ø26Ø LOCATE 2Ø, 1
[B17] 1Ø27Ø PRINT CHR$(218); STRING$(78, 196); CHR$(191
      );
[ØØ7] 1Ø28Ø PRINT CHR$(179); STRING$(78, 32); CHR$(179)
      ;
[C27] 1Ø29Ø PRINT CHR$(192); STRING$(78, 196); CHR$(217
      );
[4V1] 1Ø3ØØ COLOR 7
[9F1] 1Ø31Ø Y = 7
[9V6] 1Ø32Ø LOCATE 7, 31: PRINT "SCheck  Version 1.x"
[PR6] 1Ø33Ø LOCATE 8, 31: PRINT "SCheck  Version 2.Ø"
```

```
[SSG] 1Ø34Ø LOCATE 21, 5: PRINT "Bitte wählen Sie mit d
      en Pfeiltasten aus und bestätigen durch <RETURN>
      !";
[2PD] 1Ø35Ø COLOR 14, 1: FOR Q = 3Ø TO 5Ø: LOCATE Y, Q:
       PRINT CHR$(SCREEN(Y, Q)): NEXT Q: COLOR 7, Ø
[UP6] 1Ø36Ø TASTE$ = INKEY$: IF TASTE$ = "" THEN 1Ø36Ø
[P96] 1Ø37Ø IF TASTE$ <> CHR$(Ø) + CHR$(72) THEN 1Ø4ØØ
[4ND] 1Ø38Ø IF Y <> 7 THEN FOR Q = 3Ø TO 5Ø: LOCATE Y,
      Q: PRINT CHR$(SCREEN(Y, Q)): NEXT Q: Y = Y - 1
[532] 1Ø39Ø GOTO 1Ø35Ø
[1A6] 1Ø4ØØ IF TASTE$ <> CHR$(Ø) + CHR$(8Ø) THEN 1Ø43Ø
[1ND] 1Ø41Ø IF Y <> 8 THEN FOR Q = 3Ø TO 5Ø: LOCATE Y,
      Q: PRINT CHR$(SCREEN(Y, Q)): NEXT Q: Y = Y + 1
[232] 1Ø42Ø GOTO 1Ø35Ø
[F35] 1Ø43Ø IF TASTE$ <> CHR$(13) THEN 1Ø36Ø
[8N2] 1Ø44Ø FOR Q = 7 TO 8
[U16] 1Ø45Ø   LOCATE Q, 3Ø: PRINT STRING$(21, 32)
[KN1] 1Ø46Ø NEXT Q
[8Y2] 1Ø47Ø VERSION = Y - 6
[GV1] 1Ø48Ø COLOR 7
[NF1] 1Ø49Ø Y = 7
[MA5] 1Ø5ØØ LOCATE 7, 36: PRINT "Standard"
[4A6] 1Ø51Ø LOCATE 8, 34: PRINT "Mit Prüfsumme"
[C65] 1Ø52Ø LOCATE 9, 37: PRINT "Hexdump"
[ZSG] 1Ø53Ø LOCATE 21, 5: PRINT "Bitte wählen Sie mit d
      en Pfeiltasten aus und bestätigen durch <RETURN>
      !";
[9PD] 1Ø54Ø COLOR 14, 1: FOR Q = 3Ø TO 5Ø: LOCATE Y, Q:
       PRINT CHR$(SCREEN(Y, Q)): NEXT Q: COLOR 7, Ø
[3R6] 1Ø55Ø TASTE$ = INKEY$: IF TASTE$ = "" THEN 1Ø55Ø
[4B6] 1Ø56Ø IF TASTE$ <> CHR$(Ø) + CHR$(72) THEN 1Ø59Ø
[BND] 1Ø57Ø IF Y <> 7 THEN FOR Q = 3Ø TO 5Ø: LOCATE Y,
      Q: PRINT CHR$(SCREEN(Y, Q)): NEXT Q: Y = Y - 1
[E32] 1Ø58Ø GOTO 1Ø54Ø
[GA6] 1Ø59Ø IF TASTE$ <> CHR$(Ø) + CHR$(8Ø) THEN 1Ø62Ø
[CND] 1Ø6ØØ IF Y <> 9 THEN FOR Q = 3Ø TO 5Ø: LOCATE Y,
      Q: PRINT CHR$(SCREEN(Y, Q)): NEXT Q: Y = Y + 1
[B32] 1Ø61Ø GOTO 1Ø54Ø
[P35] 1Ø62Ø IF TASTE$ <> CHR$(13) THEN 1Ø55Ø
[GN2] 1Ø63Ø FOR Q = 7 TO 9
[126] 1Ø64Ø   LOCATE Q, 3Ø: PRINT STRING$(21, 32)
[TN1] 1Ø65Ø NEXT Q
[WG4] 1Ø66Ø IF Y = 8 THEN LAUF = TRUE
[NA4] 1Ø67Ø IF Y = 9 THEN BIN = TRUE
[RU5] 1Ø68Ø LOCATE 12, 54: PRINT STRING$(25, 32)
[C9F] 1Ø69Ø GOSUB 1315Ø: LOCATE 21, 12: PRINT "Geben Si
      e bitte den Dateinamen der gewünschten Datei an !
      ";
[9P8] 1Ø7ØØ Y = 12: X = 54: LAENGE = 24: MENGE$ = "alle
      ": GOSUB 1Ø89Ø
[N6D] 1Ø71Ø IF (TASTE$ = CHR$(27)) OR (Q$ = "") THEN LO
      CATE 12, 54: PRINT STRING$(25, 32): GOTO 1Ø68Ø
[LM3] 1Ø72Ø ON ERROR GOTO 1116Ø
[C74] 1Ø73Ø OPEN Q$ FOR INPUT AS #T
[732] 1Ø74Ø GOTO 1123Ø
[3Ø2] 1Ø75Ø COLOR 12
[2A6] 1Ø76Ø LOCATE 8, 3Ø: PRINT "D A T E I F E H L E R"
[VV1] 1Ø77Ø COLOR 7
```

Listing. »SCheck.bas« für GW-Basic

```
[RØC] 1Ø78Ø GOSUB 1315Ø: LOCATE 21, 23: PRINT "Drücken
      Sie bitte irgendeine Taste !";
[B83] 1Ø79Ø WHILE INKEY$ = ""
[GX2] 1Ø8ØØ   PRINT CHR$(7);
[CE1] 1Ø81Ø WEND
[6U5] 1Ø82Ø LOCATE 8, 3Ø: PRINT STRING$(21, 32)
[842] 1Ø83Ø GOTO 1Ø68Ø
[2VØ] 1Ø84Ø '
[184] 1Ø85Ø '*********************************
[UD6] 1Ø86Ø '* Unterprogramm zur Dateneingabe *
[384] 1Ø87Ø '*********************************
[6VØ] 1Ø88Ø '
[GG1] 1Ø89Ø Q$ = ""
[EF2] 1Ø9ØØ LOCATE Y, X
[FS3] 1Ø91Ø PRINT Q$; CHR$(219);
[GR6] 1Ø92Ø TASTE$ = INKEY$: IF TASTE$ = "" THEN 1Ø92Ø
[ZK5] 1Ø93Ø IF (TASTE$ <> CHR$(8)) THEN 1Ø99Ø
[224] 1Ø94Ø   IF LEN(Q$) = Ø THEN 1Ø92Ø
[X44] 1Ø95Ø   Q$ = LEFT$(Q$, LEN(Q$) - 1)
[6N2] 1Ø96Ø   LOCATE Y, X
[V44] 1Ø97Ø   PRINT Q$; CHR$(219); " ";
[8A2] 1Ø98Ø   GOTO 1Ø92Ø
[T98] 1Ø99Ø IF TASTE$ = CHR$(13) THEN LOCATE Y, X: PRIN
      T Q$; " ";
[639] 11ØØØ IF (TASTE$ = CHR$(13)) OR (TASTE$ = CHR$(27
      )) THEN RETURN
[C55] 11Ø1Ø IF (ASC(TASTE$) < 32) THEN 1Ø92Ø
[NX4] 11Ø2Ø IF MENGE$ <> "zahl" THEN 11Ø5Ø
[ZC8] 11Ø3Ø   IF (ASC(TASTE$) < 48) OR (ASC(TASTE$) > 5
      7) THEN 1Ø92Ø
[892] 11Ø4Ø   GOTO 11Ø7Ø
[PU4] 11Ø5Ø IF MENGE$ = "alle" THEN 11Ø7Ø
[E86] 11Ø6Ø IF INSTR(MENGE$, TASTE$) = Ø THEN 1Ø92Ø
[YF6] 11Ø7Ø IF LEN(Q$) < LAENGE THEN Q$ = Q$ + TASTE$
[T22] 11Ø8Ø GOTO 1Ø9ØØ
[6H8] 11Ø9Ø Y = 1Ø: X = 65: LAENGE = 5: MENGE$ = "zahl"
      : GOSUB 1Ø89Ø
[BV5] 111ØØ IF (TASTE$ = CHR$(27)) THEN RETURN
[6UØ] 1111Ø '
[NK2] 1112Ø '******************
[TR3] 1113Ø '* Fehler-Routine *
[RK2] 1114Ø '******************
[AUØ] 1115Ø '
[KN1] 1116Ø CLOSE
[ZG2] 1117Ø RESUME 1Ø75Ø
[DUØ] 1118Ø '
[8G2] 1119Ø '*****************
[TH3] 112ØØ '* Hauptprogramm *
[4G2] 1121Ø '*****************
[BUØ] 1122Ø '
[KP3] 1123Ø COLOR 7: GOSUB 1315Ø
[N43] 1124Ø ON ERROR GOTO Ø
[K62] 1125Ø DATEI$ = Q$
[DF7] 1126Ø GROSS$ = DATEI$: GOSUB 13Ø4Ø: DATEI$ = GROS
      S$
[3W3] 1127Ø I = INSTR(DATEI$, ".")
[6K7] 1128Ø IF I = Ø THEN DATEI$ = DATEI$ + ".": I = LE
      N(DATEI$)
[U64] 1129Ø EXT$ = MID$(DATEI$, I, 4)
[8T4] 113ØØ DATEI$ = LEFT$(DATEI$, I - 1)
[HN5] 1131Ø LOCATE 12, 54: PRINT DATEI$; EXT$
[HB4] 1132Ø TEMPNAME$ = DATEI$ + ".$$$"
[5E6] 1133Ø OPEN TEMPNAME$ FOR OUTPUT AS #TEMP
[CZ7] 1134Ø IF BIN THEN OPEN DATEI$ + ".BIN" FOR OUTPUT
       AS #F
[CP7] 1135Ø IF VERSION = 2 THEN NUMMER = 1 ELSE NUMMER
      = Ø
[KYC] 1136Ø IF BIN THEN GOSUB 1315Ø: LOCATE 21, 25: PRI
      NT ">>> Binärdatei wird erzeugt <<<"
[TE3] 1137Ø WHILE NOT EOF(T)
[X24] 1138Ø   LINE INPUT #T, ZEILE$
[LDB] 11385   WHILE RIGHT$(ZEILE$,1)=" " : ZEILE$=LEFT$
      (ZEILE$,LEN(ZEILE$)-1) : WEND
[333] 1139Ø   WEITER = FALSE
[WT3] 114ØØ   IF WEITER THEN 115ØØ
[WN6] 1141Ø   GOSUB 1182Ø 'Gelesene Zeile anzeigen
[TS4] 1142Ø   IF BIN THEN WEITER = TRUE
[ZT3] 1143Ø   IF WEITER THEN 115ØØ
[AM6] 1144Ø   CH$ = INKEY$: IF CH$ = "" THEN GOTO 1144Ø
[3G3] 1145Ø   ERLAUBT = FALSE
[ZH8] 1146Ø   IF CH$ = CHR$(13) THEN ERLAUBT = TRUE: WE
      ITER = TRUE
[N1C] 1147Ø   IF (CH$ = "e") OR (CH$ = "E") THEN ERLAUB
      T = TRUE: GOSUB 12Ø6Ø'Zeile bearbeiten
[8ME] 1148Ø   IF (CH$ = "q") OR (CH$ = "Q") THEN IF NOT
       LAUF THEN GOSUB 1242Ø: IF SCHLUSS THEN GOTO 1163
      Ø'Quit
[F57] 1149Ø   IF ERLAUBT THEN GOTO 114ØØ ELSE GOTO 1144
      Ø
[8Y3] 115ØØ   PRINT #TEMP, ZEILE$
[DU3] 1151Ø   IF NOT BIN THEN 1161Ø
[D99] 1152Ø   WHILE LEFT$(ZEILE$, 1) = " ": ZEILE$ = MI
      D$(ZEILE$, 2): WEND
[364] 1153Ø   IF ZEILE$ = "" THEN 1161Ø
[K75] 1154Ø   B = VAL("&H" + LEFT$(ZEILE$, 2))
```

```
[TSH] 1155Ø   IF (B < Ø) OR (B > 255) THEN GOSUB 1315Ø:
       LOCATE 21, 22: PRINT "+++++ ABBRUCH WEGEN DATENF
      EHLER +++++": LOCATE 25, 1: END
[HN1] 1156Ø   BY = B
[TK3] 1157Ø   PRINT #F, CHR$(BY);
[YA4] 1158Ø   ZEILE$ = MID$(ZEILE$, 3)
[M99] 1159Ø   WHILE LEFT$(ZEILE$, 1) = " ": ZEILE$ = MI
      D$(ZEILE$, 2): WEND
[T92] 116ØØ   GOTO 1153Ø
[DY3] 1161Ø   NUMMER = NUMMER + 1
[8E1] 1162Ø WEND
[S7B] 1163Ø WHILE NOT EOF(T): LINE INPUT #T, ZEILE$: PR
      INT #TEMP, ZEILE$: WEND
[4P1] 1164Ø CLOSE
[8U5] 1165Ø OPEN DATEI$ + EXT$ FOR INPUT AS #1
[XØ6] 1166Ø OPEN DATEI$ + ".BAK" FOR OUTPUT AS #2
[XGB] 1167Ø WHILE NOT EOF(1): LINE INPUT #1, ZEILE$: PR
      INT #2, ZEILE$: WEND: CLOSE
[6D5] 1168Ø OPEN TEMPNAME$ FOR INPUT AS #1
[576] 1169Ø OPEN DATEI$ + EXT$ FOR OUTPUT AS #2
[UGB] 117ØØ WHILE NOT EOF(1): LINE INPUT #1, ZEILE$: PR
      INT #2, ZEILE$: WEND: CLOSE
[713] 1171Ø KILL TEMPNAME$
[GTN] 1172Ø GOSUB 1256Ø: LOCATE 12, 9: PRINT STRING$(11
      , 32): LOCATE 12, 25: PRINT STRING$(21, 32): LOCA
      TE 12, 53: PRINT STRING$(27, 32)'Statuszeile lösc
      hen
[494] 1173Ø IF BIN THEN EXT$ = ".BIN"
[N4H] 1174Ø GOSUB 1315Ø: LOCATE 21, (8Ø - (31 + LEN(DAT
      EI$ + EXT$))) / 2: PRINT "Die Datei '"; DATEI$ +
      EXT$; "' wurde gespeichert."
[CE2] 1175Ø LOCATE 25, 1
[H51] 1176Ø END
[4VØ] 1177Ø '
[TH3] 1178Ø '***************************
[B65] 1179Ø '* Gelesene Zeile anzeigen *
[NH3] 118ØØ '***************************
[2VØ] 1181Ø '
[BBB] 1182Ø IF LAUF THEN CHECK$ = "[" + LEFT$(ZEILE$, 3
      ) + "]": ZEILE$ = MID$(ZEILE$, 5)
[MDN] 1183Ø FOR Q = 1 TO LEN(CHECK$): IF ASC(MID$(CHECK
      $, Q, 1)) > 96 AND ASC(MID$(CHECK$, Q, 1)) < 123
      THEN MID$(CHECK$, Q, 1) = CHR$(ASC(MID$(CHECK$, Q
      , 1)) - 32)
[3P1] 1184Ø NEXT Q
[B57] 1185Ø LINKS = 2: OBEN = 6: RECHTS = 79: UNTEN = 1
      Ø
[CX4] 1186Ø LOCATE 12, 26: PRINT NUMMER
[Z45] 1187Ø GOSUB 1256Ø'Fenster löschen
[WX8] 1188Ø GOSUB 1266Ø'Zeile innerhalb des Fensters an
      zeigen
[366] 1189Ø GOSUB 128ØØ'Check-Summe errechnen
[ØM2] 119ØØ LOCATE 12, 1Ø
[HB2] 1191Ø PRINT PS$
[HM7] 1192Ø IF BIN THEN GOTO 12ØØØ'Binärdatei erzeugen
[N8C] 1193Ø IF LAUF THEN GOTO 1197Ø'Übereinstimmung der
       Prüfsummen feststellen
[X74] 1194Ø GOSUB 1315Ø: LOCATE 21, 4
[LKD] 1195Ø PRINT "[E] Editieren    [RETURN] Weiter
      [Q] Quit       [Ctrl-Break] Abbrechen"
[152] 1196Ø GOTO 1199Ø
[ATP] 1197Ø GOSUB 1315Ø: IF PS$ = CHECK$ THEN LOCATE 21
      , (8Ø - (22 + LEN(STR$(NUMMER)))) / 2: PRINT "Prü
      fsumme OK für Zeile"; NUMMER: WEITER = TRUE: GOTO
       12ØØØ
[1NH] 1198Ø LOCATE 21, 4: PRINT CHR$(7); "Prüfsummenfeh
      ler!  [E] Editieren  [RETURN] Weiter  [Ctrl-Break
      ] Abbrechen"
[XM2] 1199Ø LOCATE 12, 15
[KV1] 12ØØØ RETURN
[5UØ] 12Ø1Ø '
[MK2] 12Ø2Ø '******************
[4H3] 12Ø3Ø '* Zeile eingeben *
[PK2] 12Ø4Ø '******************
[9UØ] 12Ø5Ø '
[FVD] 12Ø6Ø GOSUB 1315Ø: LOCATE 21, 17: PRINT "Ändern S
      ie die Programmzeile !     [RETURN] Ende"
[245] 12Ø7Ø GOSUB 1256Ø'Fenster löschen
[7B5] 12Ø8Ø GOSUB 1266Ø'Zeile darstellen
[FL3] 12Ø9Ø LOCATE OBEN, LINKS
[7G3] 121ØØ X = 1: PRINT "■";
[8E2] 1211Ø CH$ = INKEY$
[FE6] 1212Ø IF LEFT$(CH$, 1) <> CHR$(Ø) THEN GOTO 1222Ø
[4N3] 1213Ø   CH$ = RIGHT$(CH$, 1)
[4NE] 1214Ø   IF CH$ = "M" THEN IF X > LEN(ZEILE$) THEN
       PRINT CHR$(7); : GOTO 1211Ø ELSE GOSUB 1234Ø: X
      = X + 1
[KZC] 1215Ø   IF CH$ = "K" THEN IF X = 1 THEN PRINT CHR
      $(7); : GOTO 1211Ø ELSE GOSUB 1234Ø: X = X - 1
[YZC] 1216Ø   IF CH$ = "H" THEN IF X < 79 THEN PRINT CH
      R$(7); : GOTO 1211Ø ELSE GOSUB 1234Ø: X = X - 78
[YCF] 1217Ø   IF CH$ = "P" THEN IF X + 77 > LEN(ZEILE$)
       THEN PRINT CHR$(7); : GOTO 1211Ø ELSE GOSUB 1234
      Ø: X = X + 78
```

```
[TV7] 12180    IF CH$ = "G" THEN IF X <> 1 THEN GOSUB 12
      340: X = 1
[41C] 12190    IF CH$ = "O" THEN IF X <> LEN(ZEILE$) + 1
       THEN GOSUB 12340: X = LEN(ZEILE$) + 1
[8PM] 12200    IF CH$ = "S" THEN IF X <= LEN(ZEILE$) THE
      N ZEILE$ = LEFT$(ZEILE$, X - 1) + MID$(ZEILE$, X
      + 1): GOSUB 12660 ELSE PRINT CHR$(7); : GOTO 1211
      0
[G92] 12210    GOTO 12240
[XXN] 12220 IF CH$ = CHR$(8) THEN IF X > 1 THEN ZEILE$
      = LEFT$(ZEILE$, X - 2) + MID$(ZEILE$, X): GOSUB 1
      2660: GOSUB 12340: X = X - 1 ELSE PRINT CHR$(7);
      : GOTO 12110
[LZN] 12230 IF CH$ >= " " THEN IF LEN(ZEILE$) = 255 THE
      N PRINT CHR$(7); : GOTO 12110 ELSE ZEILE$ = LEFT$
      (ZEILE$, X - 1) + CH$ + MID$(ZEILE$, X): GOSUB 12
      660: X = X + 1
[7YB] 12240 LOCATE OBEN + INT((X - 1) / 78), LINKS - 1
      + X - INT((X - 1) / 78) * 78: PRINT "
[LK5] 12250 IF CH$ <> CHR$(13) THEN GOTO 12110
[648] 12260 IF LAUF THEN ZEILE$ = MID$(CHECK$, 2, 3) +
      " " + ZEILE$
[XL6] 12270 GOSUB 12340: GOSUB 12660'Zeile anzeigen
[2W1] 12280 RETURN
[NU0] 12290 '
[CP3] 12300 '****************************
[2H5] 12310 '* Zeichen wiederherstellen *
[EP3] 12320 '****************************
[LU0] 12330 '
[UM9] 12340 LOCATE OBEN + INT((X - 1) / 78), LINKS - 1
      + X - INT((X - 1) / 78) * 78
[YXA] 12350 IF MID$(ZEILE$, X, 1) = "" THEN PRINT " " E
      LSE PRINT MID$(ZEILE$, X, 1)
[4W1] 12360 RETURN
[RU0] 12370 '
[0A2] 12380 '***************
[3X2] 12390 '* Quit (Ende) *
[W92] 12400 '***************
[NU0] 12410 '
[U63] 12420 SCHLUSS = FALSE
[KYB] 12430 SICHER$ = "": FOR Q = 2 TO 79: SICHER$ = SI
      CHER$ + CHR$(SCREEN(21, Q)): NEXT Q
[7LD] 12440 GOSUB 13150: LOCATE 21, 16: PRINT "Drücken
      Sie bitte [J], um die Arbeit zu beenden !"
[GA6] 12450 CH$ = INKEY$: IF CH$ = "" THEN GOTO 12450
[R3B] 12460 IF (CH$ = "j") OR (CH$ = "J") THEN SCHLUSS
      = TRUE: PRINT #TEMP, ZEILE$
[BA2] 12470 GOSUB 13150
[P78] 12480 IF NOT SCHLUSS THEN LOCATE 21, 2: PRINT SIC
      HER$
[BW1] 12490 RETURN
[651] 12500 END
[TU0] 12510 '
[6R2] 12520 '******************
[U24] 12530 '* Fenster löschen *
[8R2] 12540 '******************
[XU0] 12550 '
[044] 12560 FOR Q = OBEN TO UNTEN
[Z53] 12570 LOCATE Q, LINKS
[8B6] 12580    PRINT STRING$(RECHTS - LINKS + 1, 32)
[ZN1] 12590 NEXT Q
[AW1] 12600 RETURN
[XU0] 12610 '
[265] 12620 '*****************************************
      *
[7W7] 12630 '* Inhalt von "zeile$" im Fenster anzeigen
      *
[465] 12640 '*****************************************
      *
[1V0] 12650 '
```

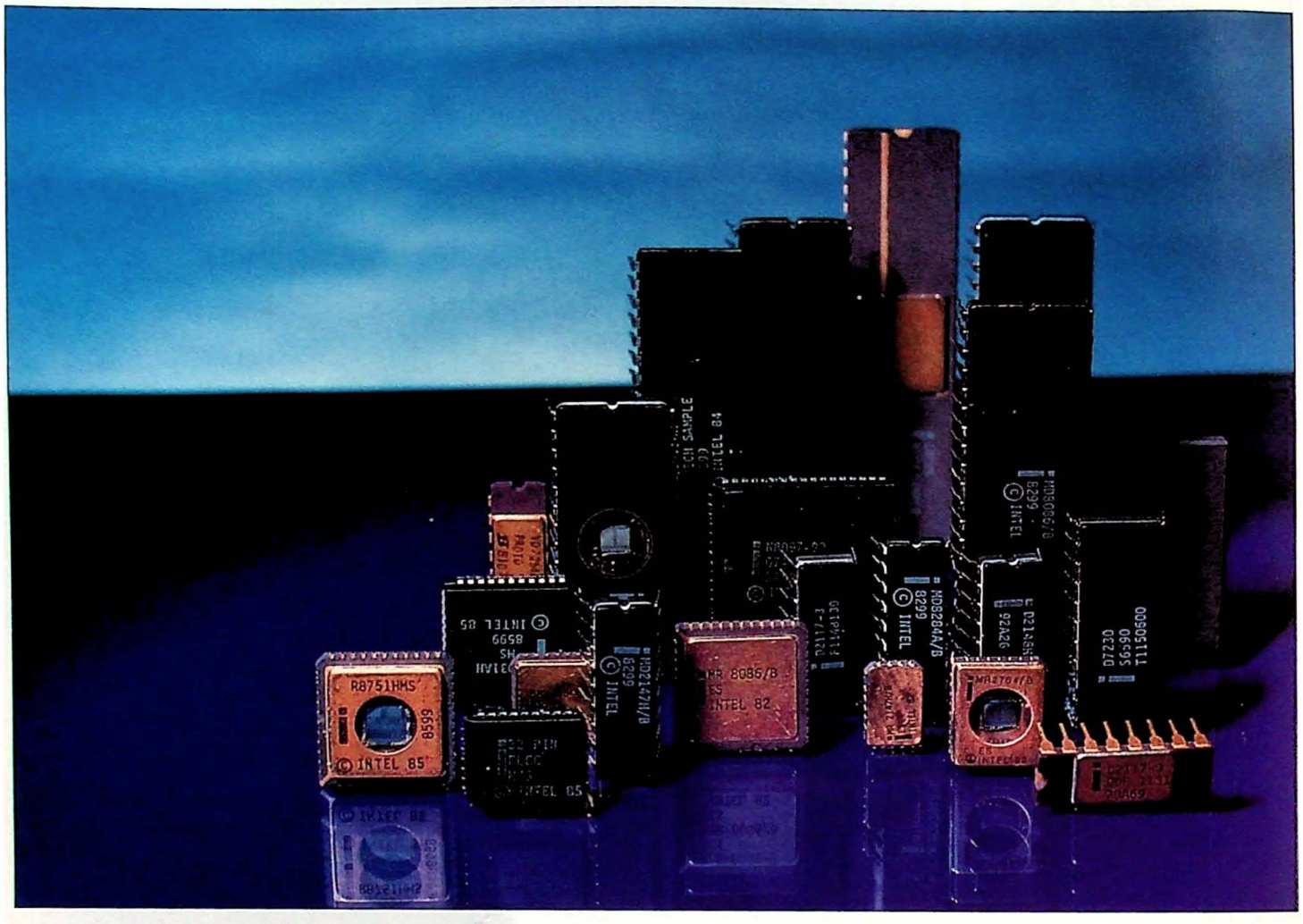
```
[SR2] 12660 F.X = 0: F.Y = 0
[CC5] 12670 FOR ZAEHLER = 1 TO LEN(ZEILE$)
[F85] 12680    LOCATE OBEN + F.Y, LINKS + F.X
[UH5] 12690    PRINT MID$(ZEILE$, ZAEHLER, 1);
[SK2] 12700    F.X = F.X + 1
[597] 12710    IF F.X >= RECHTS - 1 THEN F.X = 0: F.Y =
      F.Y + 1
[6S2] 12720 NEXT ZAEHLER
[RX6] 12730 LOCATE OBEN + F.Y, LINKS + F.X: PRINT " ";
[KW1] 12740 RETURN
[5V0] 12750 '
[AH4] 12760 '************************************
[ET6] 12770 '* Zeile auf Check-Summe überprüfen *
[CH4] 12780 '************************************
[9V0] 12790 '
[F42] 12800 P = 0: A = 1
[8SB] 12810 WHILE (MID$(ZEILE$, A, 1) = " ") OR (MID$(Z
      EILE$, A, 1) = CHR$(9)): A = A + 1: WEND
[374] 12820 FOR I = A TO LEN(ZEILE$)
[4CX] 12830    IF MID$(ZEILE$, I, 1) >= " " THEN GROSS$
      = MID$(ZEILE$, I, 1): IF VERSION = 1 THEN P = ((I
       - A + 1) AND 7) * ASC(GROSS$) + P: GOTO 12920 EL
      SE GOSUB 13040: P = (((I - A + 1) AND 3) + 1) * A
      SC(GROSS$) + P: GOTO 12920
[S22] 12840    COLOR 12
[5N6] 12850    LOCATE 8, 30: PRINT "D A T E I F E H L E
      R"
[022] 12860    COLOR 7
[K2C] 12870    LOCATE 21, 13: PRINT "          Drücken S
      ie bitte irgendeine Taste !          ";
[CC3] 12880    WHILE INKEY$ = ""
[D93] 12890      PRINT CHR$(7);
[5L1] 12900    WEND
[R16] 12910    LOCATE 8, 30: PRINT STRING$(21, 32)
[0P1] 12920 NEXT I
[KF2] 12930 PS$ = "[    ]"
[AM2] 12940 FOR I = 2 TO 4
[K25] 12945    IF P > 32767 THEN P = P - 32768
[5T7] 12950    MID$(PS$, I, 1) = MID$(ZEICHEN$, (P AND 3
      1) + 1, 1)
[HZ2] 12960    P = FIX(P / 32)
[5P1] 12970 NEXT I
[YW1] 12980 RETURN
[HV0] 12990 '
[ZX3] 13000 '*******************************
[K06] 13010 '* In Großbuchstaben umwandeln *
[934] 13020 '*******************************
[AU0] 13030 '
[9ZN] 13040 FOR Q = 1 TO LEN(GROSS$): IF ASC(MID$(GROSS
      $, Q, 1)) > 96 AND ASC(MID$(GROSS$, Q, 1)) < 123
      THEN MID$(GROSS$, Q, 1) = CHR$(ASC(MID$(GROSS$, Q
      , 1)) - 32)
[8T8] 13050 IF MID$(GROSS$, Q, 1) = "ä" THEN MID$(GROSS
      $, Q, 1) = "Ä"
[NU8] 13060 IF MID$(GROSS$, Q, 1) = "ö" THEN MID$(GROSS
      $, Q, 1) = "Ö"
[UU8] 13070 IF MID$(GROSS$, Q, 1) = "ü" THEN MID$(GROSS
      $, Q, 1) = "Ü"
[DN1] 13080 NEXT Q
[YV1] 13090 RETURN
[BU0] 13100 '
[YZ2] 13110 '**********************
[9N4] 13120 '* Meldezeile löschen *
[003] 13130 '**********************
[FU0] 13140 '
[ZT5] 13150 LOCATE 21, 2: PRINT STRING$(78, 32)
[ZV1] 13160 RETURN
```

Listing. »SCheck.bas« für GW-Basic

Utilities gesucht

Programme, die den Bildschirminhalt auf den Kopf stellen, suchen unsere Leser sicherlich nicht. Aber einen wissenschaftlichen Taschenrechner, ein Festplatten-Backup-Programm, einen Extended-Memory-Treiber, Druckerspooler oder Cache-Speicher. Zögern Sie deshalb nicht, schicken Sie uns unverbindlich die aktuelle Version Ihres Programms und einen Ausdruck der Dokumentation.
Falls Sie sich vorweg informieren möchten: Zwischen 10.00 und 12.00 Uhr und zwischen 15.00 und 17.00 Uhr stehen wir Ihnen an allen Werktagen unter der Rufnummer (089) 4 61 00 10 zur Verfügung.

Schicken Sie Ihre Unterlagen an folgende Adresse:
Redaktionsbüro Everts & Hagedorn
»DOS Software«
Hans-Stießberger-Str. 3
8013 Haar/München



Assembler zum ersten...

Maschinensprache ist die schnellste aller Programmiersprachen, weil der Computer sie direkt versteht. Auch läßt sich mit Maschinensprache ohne Einschränkungen direkt auf die Hardware eines Computers zugreifen, was bei anderen Programmiersprachen nicht oder nur bedingt der Fall ist. Aus diesem Grund sind Maschinenspracheprogramme besonders leistungsfähig.

Menschen verständigen sich auf die gleiche Weise mit Computern, wie sie auch untereinander kommunizieren: durch eine vereinbarte Sprache. Wenn zwei Kommunikationspartner die gleiche Sprache benutzen, so stellt die Verständigung kein Problem dar. Sprechen sie aber unterschiedliche Sprachen, so benötigen sie einen Dolmetscher, der die Worte des einen Gesprächspartners in die Sprache des anderen übersetzt. Es leuchtet ein, daß das Gespräch über den Dolmetscher viel Aufwand und Zeit kostet.

Außerdem ist es nicht ausgeschlossen, daß sich aufgrund der Übersetzung Mißverständnisse ergeben, die bei der direkten Kommunikation nicht auftreten würden. Selbst unterschiedliche Dialekte innerhalb derselben Sprache (zum Beispiel Plattdeutsch und Bayrisch) führen oft schon zu unüberbrückbaren Sprachbarrieren.

Auf diesem Kommunikationsprinzip basiert auch die Verständigung mit dem Computer. Die Mehrheit der Programmierer – zu der Sie wahrscheinlich auch gehören – verwendet eine höhere Programmiersprache wie Basic oder Pascal. Jede dieser Hochsprachen übersetzt das als Quelltext vorliegende Programm in einen Code, den der Computer unmittelbar verstehen und ausführen kann: die Maschinensprache. Bei den höheren Programmiersprachen gibt es zwei verschiedene Verfahren, um Hochsprachenbefehle in Maschinensprache zu übersetzen, und zwar per Interpreter oder Compiler. Ein Interpreter übersetzt ein Hochsprachenprogramm während der Ausführung Befehl für Befehl in Maschinensprache. Weil jedoch beim Ausführen eines Befehls neben der Übersetzung und den gesamten Fehlerabfangroutinen auch noch die vom jeweiligen Kommando benötigten Systemroutinen aufgerufen werden müssen, ist die erreichbare Verarbeitungsgeschwindigkeit in den meisten Fällen vergleichsweise gering.

Ein Compiler hingegen übersetzt einen Quelltext nach erfolgreicher Fehlerbeseitigung in einen maschinenspracheähnlichen Code (Objectcode) oder sogar direkt in Maschinensprache. Das auf diese Weise übersetzte Programm erreicht natürlich eine wesentlich höhere Geschwindigkeit als ein Interpreter.

Ein Maschinenspracheprogramm besteht aus einzelnen Maschinensprachebefehlen, die der Prozessor des Computers direkt ausführen kann. In einem PC arbeitet entweder der Intel-Prozessor 8086 oder 8088 und in ei-

nem AT der abwärtskompatible 80286-Prozessor, der sich im vom MS-DOS gewählten Real-Modus wie ein 8086 verhält.
Der Unterschied zwischen 8088 und 8086 besteht darin, daß der 8088 im Gegensatz zum 8086, der einen 16-Bit-Datenbus besitzt, nur über einen 8 Bit breiten Datenbus verfügt. Dies vereinfacht zwar den elektrischen Aufbau eines PC, doch ist der 8088 dadurch etwas weniger leistungsfähig als der 8086, weil beim 8088 16-Bit-Daten nicht parallel übertragen werden können. Sie müssen hierbei in zwei 8-Bit-Einheiten hintereinander auf die »Datenreise« geschickt werden.
Der 8088 besitzt auch nur einen 4-Byte-Befehlspuffer (»Instruction Fetch Queue«) für Maschinensprachebefehle, während der Befehlspuffer beim 8086 sechs Byte lang ist, so daß dieser Prozessor »vorausschauender« arbeiten kann.

Maschinensprache kann der Prozessor des Computers ohne Übersetzung verstehen

Der Befehlssatz des 8086/8088 umfaßt 120 Maschinensprachebefehle, die mit keinem, einem oder zwei Operanden arbeiten. Jeder dieser Befehle wird in der vom Prozessor ausführbaren Form durch einen numerischen Wert – den Maschinencode – repräsentiert. Der Maschinencode kann je nach Komplexität des Befehls ein oder zwei Byte lang sein.
Operanden, die für arithmetische, logische oder sonstige Operationen benötigt werden, sind in Maschinenspracheprogrammen ebenfalls als Zahlen abgelegt. Auch die Operanden können ein oder zwei Byte lang sein. Weil ein Befehl maximal zwei Operanden haben darf, reicht damit die Länge der Maschinensprachebefehle des 8086/8088 von ein bis sechs Byte.
Befehle und Operanden werden zwar gemeinsam im Programm als Zahlen abgelegt, aber der Prozessor kann dennoch beide voneinander unterscheiden, weil er die erste Zahl immer als Maschinencode für einen Befehl behandelt. Der Maschinencode gibt an, ob kein, ein oder zwei Operanden folgen. Dadurch erkennt der Prozessor auch, welche Zahl den nächsten Maschinencode bildet.
So sind zum Beispiel die beiden aufeinanderfolgenden Zahlen 210 und 0, die in zwei hintereinanderliegenden Speicherstellen abgelegt sind, gültige Maschinensprachebefehle. Sie weisen den Prozessor an, in das Register AH den Wert 0 zu laden. Ein Register ist ein schneller Zwischenspeicher, der sich direkt im Prozessor befindet und damit einen zeitaufwendigen Zugriff auf den Arbeitsspeicher überflüssig macht.

Assemblerbefehle statt Maschinencode erleichtern die Programmierung in Maschinensprache

Weil sich in den Zahlenkolonnen von längeren Maschinenspracheprogrammen auf die Dauer niemand zurechtfindet, hat sich der Hersteller Intel für seine Prozessoren sinnvolle Kurzbezeichnungen einfallen lassen, die anstelle der Zahlenfolgen verwendet werden. Diese Kurzbezeichnungen bezeichnet man als Assemblerbefehle oder Opcodes (für »Operation Codes«).
Um den 8086/8088 zu veranlassen, den Wert 0 in das AH-Register zu schreiben, gibt der Programmierer somit nicht mehr die Zahlen 210 und 0 an, sondern schreibt statt dessen

```
mov ah,0
```

Die Kurzbezeichnung »mov« steht für das englische Wort »move« und bedeutet »bewegen«. Der Assemblerbefehl »mov ah,0« bedeutet also dem Prozessor, den Wert 0 in das Register AH zu bewegen.
Sie wenden jetzt vielleicht ein, daß durch die Assemblerbefehle doch wieder ein Übersetzungsvorgang nötig ist, nämlich von Assemblersprache in Maschinensprache. Sie haben recht, doch im Gegensatz zu Hochsprachenbefehlen lassen sich die Assemblerbefehle 1:1 in Maschinensprache übersetzen, so daß keinerlei Übersetzungsverluste auftreten.
Ein Programm, das die symbolischen Assemblerbefehle in den numerischen Maschinencode übersetzt, heißt – wie sollte es anders sein – Assembler.
Die Assemblersprache – und damit natürlich auch die Maschinensprache – greift direkt auf den Speicher des Computers zu. Nun ist dieser Speicher allerdings so aufgebaut, daß er nur zwischen zwei elektrischen Zuständen unterscheiden kann. Ein Speicherbaustein kennt nur die beiden Zustände »Spannung ein« und »Spannnung aus«, die auch als logische Zustände 1 und 0 bezeichnet werden. Aus diesem Grund kommen Sie beim Erlernen der Assemblersprache nicht darum herum, sich mit einer vom Dezimalsystem abweichenden Methode der Zahlendarstellung zu beschäftigen: dem Binär- oder Dualsystem.

Die drei für die Programmierung wichtigen Zahlensysteme in der Gegenüberstellung

Dezimal	Binär	Hexadezimal
0	00000	00
1	00001	01
2	00010	02
3	00011	03
4	00100	04
5	00101	05
6	00110	06
7	00111	07
8	01000	08
9	01001	09
10	01010	0A
11	01011	0B
12	01100	0C
13	01101	0D
14	01110	0E
15	01111	0F
16	10000	10
17	10001	11
18	10010	12
19	10011	13

Die Zahl 19 setzt sich beispielsweise stellenwertig folgendermaßen zusammen:

$1x10 + 9x1 = 1x10^1 + 9x10^0$ im Zehnersystem

$1x16 + 1x2 + 1x1 = 1x2^4 + 0x2^3 + 0x2^2 + 1x2^1 + 1x2^0$ im Binärsystem

$1x16 + 3x1 = 1x16^1 + 3x16^0$ im Hexadezimalsystem

Tabelle. Die ersten 20 Zahlen des Dezimal-, Binär- und Hexadezimalsystems

Das Dezimalsystem funktioniert nach einem recht einfachen Prinzip: Jede Stelle einer Dezimalzahl kann zehn verschiedene Werte (0 bis 9) annehmen und stellt damit das Vielfache einer bestimmten Zehnerpotenz dar. Die letzte Ziffer zeigt immer das Vielfache von 10^0 an, die vorletzte Ziffer das Vielfache von 10^1 und so weiter. Die Dezimalzahl 234 setzt sich beispielsweise wie folgt zusammen:

$$
\begin{aligned}
&234\\
&= 200 + 30 + 4\\
&= 2x10^2\\
&+ 3x10^1\\
&+ 4x10^0
\end{aligned}
$$

Das Binärsystem arbeitet nach dem gleichen Prinzip, doch können hier die Stellen einer Zahl nur zwei verschiedene Werte (0 und 1) annehmen. Jede Stelle stellt somit das Vielfache einer bestimmten Zweierpotenz dar. Die letzte Ziffer zeigt immer das Vielfache von 2^0 an, die vorletzte Ziffer das Vielfache von 2^1, und so weiter.
Die Dezimalzahl 234 sieht im Binärsystem folgendermaßen aus:

234
= 128 + 64 + 32 + 8 + 2
= $1x2^7$
+ $1x2^6$
+ $1x2^5$
+ $0x2^4$
+ $1x2^3$
+ $0x2^2$
+ $1x2^1$
+ $0x2^0$
= 11101010bin

Versuchen Sie einmal zur Probe, die Zahlen 5, 36 und 158 ins Binärsystem und die Zahlen 01001001, 10101010 und 10110111 ins Dezimalsystem umzuwandeln (Auflösung am Ende).

Mit Hexadezimalzahlen lassen sich binäre Werte bequem und platzsparend zusammmenfassen

Binärzahlen haben den Vorteil, daß sich mit Ihnen die kleinste Informationseinheit – das Bit – darstellen läßt. Doch die Binärdarstellung hat auch einen Nachteil. Wenn Sie den dezimalen Wert 234 mit seinem binären Gegenwert 11101010 vergleichen, stellen Sie fest, daß Binärwerte mit zunehmender Größe auch in die Länge wachsen, so daß sie schnell »unhandlich« werden.

Abhilfe schafft das Hexadezimalsystem, das vier Binärziffern zu einer hexadezimalen Ziffer zusammenfaßt, indem jede Stelle einer Hexadezimalzahl $2^4 = 16$ verschiedene Zustände (0 bis 9 und A bis F) annehmen kann. Dadurch werden 8-Bit-Werte mit zwei und 16-Bit-Werte mit nur vier hexadezimalen Stellen dargestellt. Die dezimale Zahl 234 wird beispielsweise zu dem kompakten Wert EAhex. Die Tabelle zeigt eine Gegenüberstellung der 20 ersten Dezimal-, Binär- und Hexadezimalzahlen.

Jetzt dürfte es Ihnen auch nicht mehr schwerfallen, die Dezimalzahlen 255, 616 und 7654 in ihr hexadezimales Äquivalent umzurechnen. Zur Umrechnung von binär in hexadezimal fassen Sie einfach von hinten beginnend je vier Binärziffern zu einer hexadezimalen Ziffer zusammen. Der Binärwert

1011100

ergibt beispielsweise die Ziffern

0101 und 1100

so daß das hexadezimale Ergebnis

0101 = 5
1100 = C (12 dezimal)

5Chex lautet.

Der Aufbau des 8086/8088 bestimmt die Leistungsfähigkeit der Befehle sowie die Anzahl der Register und Adressierungsarten für Speicherzellen, die dem Programmierer beim Schreiben seiner Assemblerprogramme zur Verfügung stehen (vergleiche »Der Registersatz des 8086/80286-Prozessors« in dieser Ausgabe).

Der 8086/8088-Prozessor verfügt über einen reichhaltigen Registersatz

Der 8086/8088 besitzt die acht 16-Bit-Datenregister AX, BX, CX, DX, SI, DI, SP und BP. Die Register AX bis DX können Sie zusätzlich in je zwei 8-Bit-Register AL und AH, BL und BH, CL und CH sowie DL und DH aufteilen. Für die Segmentadressierung (eine besondere Form der Adressierung beim 8086/8088) stehen die vier 16-Bit-Segmentregister CS, DS, ES und SS bereit. Das 16-Bit-Statusregister PSW (»Processor Status Word«) zeigt den Zustand der Statusbits an und der 16-Bit-Befehlszeiger (»Instruction Pointer«) deutet auf den nächsten auszuführenden Befehl im Speicher des Computers.

Fast alle Prozessoren bearbeiten ein Programm, indem Sie einen Befehl aus dem Speicher lesen und ihn ausführen. Daraufhin wird der nächste Befehl gelesen, ausgeführt, und so geht es weiter. Diese Vorgehensweise bedingt aber einige Verzögerungen, weil der Prozessor jedesmal warten muß, bis ein neuer Befehl aus dem Speicher gelesen wird, bevor er ihn dann ausführen kann.

Der Befehlspuffer des 8086/8088 sorgt für eine beschleunigte Befehlsverarbeitung

Der 8086/8088 umgeht einen Teil dieses zeitaufwendigen Vorgangs, indem er den Aufgaben »Befehl lesen« und »Befehl ausführen« zwei separate Funktionseinheiten auf dem Baustein zuordnet: Die erste Funktionseinheit hat ausschließlich die Aufgabe, Befehle aus dem Speicher zu lesen und sie in den bereits erwähnten Befehlspuffer zu schieben. Dort stehen sie für die zweite Funktionseinheit, die die Befehle ausführt, zur Verfügung. Weil die erste Funktionseinheit immer aufeinanderfolgende Adressen liest, muß die zweite Funktionseinheit nur noch dann eingreifen und selbst einen Befehl lesen, wenn die Programmausführung durch einen Sprung an einer anderen Speicheradresse fortgeführt wird.

Alle Register des 8086/8088 sind 16 Bit breit. Mit einem 16-Bit-Wert lassen sich jedoch nur $2^{16} = 65536$ Speicherstellen adressieren (64 KByte). Dagegen beträgt der Speicherbereich des 8086/8088 aufgrund des 20 Bit breiten Datenbus 1 MByte. Wie adressiert nun der 8086/8088 aber den gesamten Speicherbereich?

Dazu benutzt er die Segmentregister, die die Startadresse (Basisadresse) für verschiedene Speichersegmente enthalten. Die 16-Bit-Adressen der normalen Register geben dagegen nur die Speicherpositionen innerhalb dieser Segmente an und werden deshalb auch als Offsets bezeichnet.

Zum Glück übernimmt bei der Programmierung in Assemblersprache der Assembler selbst die korrekte Adressierung durch die Segmentregister. Der Programmierer muß in einem Assemblerprogramm lediglich definieren, welcher Teil eines Programms zu ein und demselben Segment gehört. Die Berechnung der Adressen für die Segmentregister erledigt dann, wie gesagt, der Assembler.

Das erste Assemblerprogramm gibt einen Buchstaben auf dem Bildschirm aus

Nach so viel Theorie schreiben Sie nun Ihr erstes Assemblerprogramm. Als Hilfsmittel eignet sich der MS-DOS-Debugger »Debug«, der sich auf jeder MS-DOS-Systemdiskette befindet. Ein Debugger ist ein Programm, mit dessen Hilfe sich einzelne Speicherbereiche des Computer anzeigen und ändern lassen. »Debug« enthält sogar einen kleinen Assembler, der es Ihnen erlaubt, Maschinensprachebefehle auch in Assemblerform in den Speicher zu schreiben.

Ihr erstes Programm gibt auf dem Bildschirm den Buchstaben »A« aus. In der MS-DOS-Kommandoebene starten Sie dazu den Debugger durch die Eingabe von »debug«. Nach kurzer Zeit erscheint auf dem Bildschirm ein Minuszeichen, das Bereitschaftssymbol (Prompt) des Debuggers. Der Prompt zeigt dem Benutzer an, daß der Debugger auf eine Eingabe wartet.

Wenn Sie nun den Assemblerbefehl

a0100

eingeben, speichert »Debug« alle Assemblerbefehle, die Sie im folgenden schreiben, als ausführbaren Maschinencode ab der Offsetadresse 0100 im Arbeitsspeicher des PC.

Geben Sie darauf das folgende Assemblerlisting ein und schließen Sie

die Eingabe durch zweimaliges Betätigen der Enter-Taste ab:

```
mov dl,41
mov ah,6
int 21
mov ah,4c
int 21
```

Dieses kleine Programm lädt mit dem Ihnen bereits bekannten Mov-Befehl den ASCII-Wert von »A« (41hex) in das Register DL, schreibt in das AH-Register die Zahl 6 als Nummer für eine MS-DOS-Funktion und ruft über den Software-Interrupt 21hex die MS-DOS-Funktion 6 als Unterprogramm auf.
Die MS-DOS-Funktion 6 gibt den Wert im DL-Register als ASCII-Zeichen auf dem Bildschirm aus. Abschließend wird die MS-DOS-Funktion 4Chex über den Software-Interrupt 21hex aufgerufen, die das Programm ordnungsgemäß beendet.
MS-DOS-Funktionen und Software-Interrupts werden im Artikel »Dateitypen und MS-DOS-Funktionen« ausführlicher behandelt, doch vorerst reicht die Erklärung, daß sich mit Software-Interrupts und MS-DOS-Funktionen die Betriebssystemroutinen des PC aufrufen lassen.
Mit dem Go-Befehl

```
g=0100
```

können Sie Ihr Programm nach erfolgter Eingabe starten und überprüfen, ob es korrekt funktioniert.

Mit dem Debugger können Sie nicht nur disassemblieren

Um das Programm als ausführbare Datei auf Diskette oder Festplatte zu speichern, müssen Sie dessen Länge bestimmen. Betrachten Sie dazu mit dem Unassemble-Befehl

```
u0100
```

seinen Maschinencode im Speicher. Dieser Befehl disassembliert gleichzeitig den Maschinencode in die Assemblerbefehle zurück, so daß Sie in aller Ruhe den hexadezimalen Maschinencode mit den Assemblerbefehlen vergleichen können. Suchen Sie den zweiten Interrupt-Befehl, der auch den letzten Befehl des Programms darstellt und zählen Sie bis zu diesem Punkt die Bytes des Maschinencodes ab (jeweils zwei hexadezimale Ziffern sind ein Byte). Es müssen insgesamt 10 Byte sein (B2, 41, B4, 06, CD, 21, B4, 4C, CD und 21).
Die Länge des Programms teilen Sie dem Debugger mit, indem Sie das CX-Register auf diesen Wert setzen. Mit »r cx« rufen Sie die Wertzuweisung für das Register CX auf und geben hinter dem Doppelpunkt den hexadezimalen Wert 0Ahex ein, der dem dezimalen Wert 10 entspricht. Jetzt müssen Sie Ihrem Programm nur noch einen Namen geben. Dies geschieht mit dem Name-Befehl. Die Anweisung

```
n demo.com
```

weist beispielsweise Ihrem Programm den Namen »demo.com« zu. Wichtig ist die Erweiterung (Extension) »com«, weil sich das Programm sonst nicht korrekt starten läßt. Mit dem Write-Befehl »w« speichern Sie schließlich die Datei.

Bei längeren Maschinenspracheprogrammen ist ein Assembler dem Debugger gegenüber die bessere Wahl

Wenn Sie danach mit »q« für »quit« den Debugger verlassen, läßt sich das Programm durch die Eingabe von »demo« aus der MS-DOS-Kommandoebene starten, so daß der Buchstabe »A« unter dem MS-DOS-Prompt erscheint. Schreiben Sie einen anderen Wert in das DL-Register, so wird das entsprechende ASCII-Zeichen auf den Bildschirm ausgegeben.

Bei kurzen Maschinenspracheprogrammen eignet sich der Debugger zur Befehlseingabe durchaus noch recht gut, doch für längere Programme ist das beschriebene Verfahren zu umständlich, wenn auch nicht direkt undurchführbar. Hier ist es wesentlich einfacher, zuerst den gesamten Quelltext einzugeben, um ihn dann komplett von einem Assembler übersetzen zu lassen. In diesem Fall können Sie auch mit symbolischen Sprung- und Adreßmarken arbeiten, so daß Sie bei einer Verzweigung im Programm oder für ein Datenfeld nicht jedesmal umständlich die absolute Adresse berechnen müssen. Zudem würden sich die meisten Adressen bei jeder kleinen Programmänderung verschieben, so daß sie neu zu berechnen wären.
Ein Assembler erlaubt Ihnen dagegen, für Sprung- und Adreßmarken statt Zahlenwerten selbstgewählte Namen zu benutzen, die wesentlich aussagekräftiger und bequemer zu handhaben sind. Die Sprung- und Adreßmarken rechnet der Assembler erst beim Assemblierungsvorgang automatisch in absolute Adressen um.
Für die Eingabe eines Assemblerprogramms benötigen Sie lediglich einen Editor, der ASCII-Texte erzeugt. Dies ist beispielsweise ein Textverarbeitungsprogramm wie »Word«, das MS-DOS-Programm »Edlin« oder das Schneider-PC-Programm »Rped«. Listing 1 und 2 zeigen zwei kleine Assemblerprogramme, die über die MS-DOS-Funktion 9 einen Text auf den Bildschirm ausgeben und für den Makroassembler MASM von Microsoft (Listing 1), beziehungsweise den Assembler SAM aus dieser DOS-Extra-Ausgabe (Listing 2) angepaßt sind. Wenn Sie noch nie mit Assemblerprogrammen zu tun hatten, werden Ihnen die beiden Listings kaum etwas sagen. Deshalb finden Sie die Listings etwas später noch detailliert erklärt.
Tippen Sie jedoch zunächst einmal eines der beiden Programme (abhängig davon, welchen Assembler Sie besitzen) Buchstabe für Buchstabe ab und speichern Sie es unter dem Namen »print.asm« (für den MASM) beziehungsweise »print.sam« (für SAM) auf Diskette oder Festplatte.
Beim Übersetzen von Programmen unterscheiden Assembler zwischen zwei Arten von Befehlen: auf der einen Seite die »echten« Maschinensprachebefehle in Assemblerform (»jmp«, »mov«, »int«, etc.), auch Opcodes genannt, und auf der anderen Seite die Pseudobefehle (»title«, »assume«, »org«, etc.), auch Pseudo-Opcodes genannt.

Pseudobefehle oder Pseudo-Opcodes stellen Anweisungen für den Assembler dar

Während der Assembler die Maschinensprachebefehle direkt in den vom Prozessor ausführbaren Maschinencode umwandelt, ist ein Pseudobefehl kein Maschinensprachebefehl des Prozessors, sondern stellt vielmehr eine Anweisung für den Assembler dar. Ein Pseudobefehl bestimmt beispielsweise, von welcher Speicheradresse an ein Programm beginnt (»org«), definiert die Grenzen von Speichersegmenten (»segment« und »ends«) oder legt das logische Programmende fest (»end«).
Die Anzahl der Pseudobefehle eines Assemblers bestimmt seine Leistungsfähigkeit und den Komfort. Auf diesem Gebiet tut sich der MASM von Microsoft sehr hervor.
Nun wird, wie versprochen, Listing 1 vollständig erklärt. Weil das Programm ein typisches Assemblerlisting ist, lernen Sie dabei auch gleichzeitig die allgemeine Struktur eines Assemblerprogramms kennen. Die Erläuterungen gelten auch für die SAM-Version in Listing 2, nur daß SAM die Pseudobefehle, die die Speichersegmente betreffen, nicht benötigt.

Allgemein besteht der Text von Assemblerprogrammen aus bis zu vier Spalten. Die erste Spalte enthält die Namen von Sprungmarken (Labels) und Variablen. Den Marken muß – normalerweise – immer ein Doppelpunkt folgen. In der zweiten Spalte befinden sich die Assemblerbefehle und Pseudobefehle, und in der dritten Spalte des Programmtextes stehen die Bezeichnungen von Registern und Operanden für die Assembler- beziehungsweise Pseudobefehle. Die vierte und letzte Spalte enthält Kommentare, die durch ein Semikolon gekennzeichnet sind und die der Assembler ignoriert. Diese Kommentare dürfen auch am Anfang einer Zeile stehen, sofern sie mit einem Semikolon beginnen.

In der ersten Zeile von Listing 1 definiert der Pseudobefehl »title« den Programmnamen »print«. Darauf folgen drei Kommentarzeilen, die die Aufgabe des Programms erläutern. Mit dem Pseudobefehl »assume« wird den Segmentregistern CS und DS das (einzige) Speichersegment »code« zugeordnet. Die Segmentregister ES und SS kommen im Print-Programm nicht zur Anwendung und fehlen aus diesem Grunde.

Der Pseudobefehl »segment« definiert den Segmentstart von »code« und mit dem Pseudobefehl »org« läßt sich die Startadresse des Programms innerhalb dieses Segments festlegen. In diesem Fall wurde die Adresse 100hex gewählt, weil das Programm dadurch als Datei mit der Endung »com« lauffähig ist. Das Listing ist als Com-Datei angelegt, weil dies den Programmaufbau in Assemblersprache vereinfacht und die Programmlänge minimiert. Die Erklärung des genauen Unterschieds zwischen Dateien mit der Endung »com« und »exe« folgt im Beitrag »Dateitypen und MS-DOS-Funktionen«.

Die MS-DOS-Funktion 9 gibt Zeichenketten auf dem Bildschirm aus

»start:« ist die erste und einzige Marke im Programm. Sie bezieht sich auf den Pseudobefehl »end« und markiert den Startpunkt des Programms. Der einem End-Befehl folgende Name definiert nämlich jeweils die Sprungmarke, die beim Start eines Programms aufgerufen wird.

Der erste echte Assemblerbefehl des Programms ist der Sprungbefehl »jmp«, der beim SAM-Assembler »jmps« (für »jump short«) heißen muß. Dieser Befehl überspringt den Datenbereich im Quellcode, der den auszugebenden Text enthält und ruft den eigentlichen Anfang des Programms auf. Der Text ist im Datenbereich der Variablen »text« über den Pseudobefehl »db« (»define byte«) zugeordnet.

Selbstverständlich können Sie hier auch einen anderen Text einsetzen. Wichtig ist nur, daß der Text mit einem Dollarzeichen abschließt, das für die MS-DOS-Funktion 9 bei der Ausgabe auf dem Bildschirm die Endbegrenzung darstellt. Die dezimalen Werte 10 und 13 sind die ASCII-Zeichen für Zeilenvorschub und Wagenrücklauf. Dadurch wird vor und nach dem Text je eine Leerzeile ausgegeben.

Mit dem ersten Mov-Befehl beginnt das eigentliche Assemblerprogramm. Der Pseudobefehl »offset« bewirkt, daß die Adresse des Speicherbereichs der Variablen »text«, die den auszugebenden Text enthält, in das DX-Register geschrieben wird. Fehlt der Pseudobefehl »offset«, so wird nicht die Adresse, sondern das erste Zeichen der Variablen »text« selbst in das DX-Register geladen. Anschließend wird durch den Wert 9 im AH-Register und über den Software-Interrupt 21hex die MS-DOS-Funktion 9 aufgerufen, die die in »text« enthaltene Zeichenkette auf dem Bildschirm ausgibt. Der Wert in DX teilt der MS-DOS-Funktion 9 die Startadresse dieser Zeichenkette mit, und das Dollarzeichen kennzeichnet das Ende des Textes.

Der erste Mov-Befehl leitet das Assembler-Programm ein

Auch das Programmende führt eine MS-DOS-Funktion des Software-Interrupt 21hex herbei. Diesmal wird die Funktion 4Chex aufgerufen. Über diese Funktion kann das Programm problemlos in die MS-DOS-Kommandoebene zurückkehren. Der Pseudobefehl »ends« gibt schließlich das Ende des Segments »code« an, und der Befehl »end« markiert das Programmende.

Wenn Sie das Programm nun assemblieren möchten, rufen Sie den MASM oder SAM auf, indem Sie in der MS-DOS-Kommandoebene den Dateinamen

```
masm
```

oder »sam« eingeben und den Assembler mit [Enter] starten. Alle weiteren Schritte ergeben sich aus den Angaben, die die Assembler benötigen.

Um sich beim MASM nicht erst in dessen Bedienung einarbeiten zu müssen, schreiben Sie sich am einfachsten eine kleine Stapeldatei, die ein Assemblerprogramm automatisch übersetzt. Geben Sie dazu in der Kommandoebene folgendes ein:

```
copy con asm.bat
masm %1 ;
link %1 ;
exe2bin %1 %1.com
```

Anschließend betätigen Sie die Tasten [Enter], [F6] und nochmals [Enter]. Damit besitzen Sie eine Stapeldatei, die Ihre Assemblerlistings auf Befehl übersetzt.

Sorgen Sie dafür, daß sich die oben beschriebene Stapeldatei, der MASM, der Linker »link« des MASM, der MS-DOS-Befehl »exe2bin« und in diesem Fall das Assemblerprogramm »print.asm« auf der gleichen Diskette beziehungsweise im gleichen Unterverzeichnis befinden.

Eine kleine Stapeldatei vereinfacht den Umgang mit dem MASM erheblich

Wenn Sie das Programm »print.asm« mit »asm print.asm« übersetzen, erzeugt der MASM aus dem Assemblerprogramm zuerst ein Objektmodul, der Linker formt das Modul in eine Datei mit der Endung »exe« um, und zuletzt wandelt »exe2bin« die Exe-Datei in eine Datei mit der Endung »com« um.

Ist die Stapeldatei beendet, so finden Sie im Verzeichnis Ihrer Diskette die beiden neuen Dateien »print.exe« und »print.com«. Die Datei »print.exe« können Sie löschen, weil sie nur eine Zwischenstufe darstellt und nicht korrekt funktioniert. »print.com« ist dagegen das Endergebnis Ihrer Bemühungen und läßt sich durch Eingabe von

```
print
```

starten. Sollte die Textausgabe wider Erwarten nicht funktionieren, dann vollziehen Sie die oben beschriebenen Schritte nochmals sorgfältig nach. Haben Sie das Listing richtig abgetippt und unter dem Namen »print.asm« auf Diskette gespeichert? Stimmt der Inhalt von »asm.bat« mit den oben beschriebenen Befehlen überein? Sind alle notwendigen Dateien auf der Diskette vorhanden?

So, nun haben Sie die erste Hürde überwunden. Sicherlich ist der Einstieg in die Maschinensprachprogrammierung nicht einfach, doch mit wachsender Erfahrung fällt Ihnen die Programmierung in Assembler leichter und Sie werden überrascht sein, welch beeindruckende Ergebnisse Sie erzielen.

(Martin Althaus/Markus Zietlow/al)

```
[RF6]    1:      title   print ; Programmname »print«
[FG6]    2:  ; Dieses Programm gibt auf dem Bildschirm
[GP6]    3:  ; eine Zeichenkette aus und kehrt darauf
[W96]    4:  ; in die MS-DOS-Kommandoebene zurück
[327]    5: assume cs:code,ds:code ; Segment »code«
[HN3]    6:                        ; für Segmentregister
[ØB3]    7:                        ; CS und DS vorsehen
[2Z6]    8: code   segment         ; Segmentstart v »code«
[SP4]    9:        org  1ØØh       ; Programmstart ab
[393]   1Ø:                        ; 1ØØhex für COM-Da-
[8D3]   11:                        ; teien erforderlich
[GT6]   12: start: jmp  anfang     ; Sprung zum Programm-
[P71]   13:                        ; anfang
[ØØØ]   14:
[1Y2]   15:                     ;  -- Datenbereich --
[H59]   16: text  db 13,1Ø,"Ein Assemblerprogramm für d
          en MASM",13,1Ø,"$"
[ØØØ]   17:
[4K3]   18:                     ;  -- Programmbereich --
[CR4]   19: anfang: mov  dx,offset text
[TD3]   2Ø:                        ; Adresse des Strings
[AE5]   21:       mov  ah,Ø9h      ; String über Inter-
[S15]   22:       int  21h         ; rupt 21hex, Funktion
[XM3]   23:                        ; Ø9hex auf Bildschirm
[ML1]   24:                        ; ausgeben
[KZ5]   25:       mov  ah,4ch      ; Programm per Interrupt
[FA5]   26:       int  21h         ; 21hex, Funktion 4Chex
[8C1]   27:                        ; beenden
[TV6]   28: code ends              ; Ende von Segment »code«
[UR5]   29:       end  start       ; Programmende und Defi-
[EC4]   3Ø:                        ; nition des Startpunktes
```

Listing 1. Das Assemblerprogramm in der Version für den Makroassembler MASM von Microsoft...

```
[FG6]    1:  ; Dieses Programm gibt auf dem Bildschirm
[GP6]    2:  ; eine Zeichenkette aus und kehrt darauf
[W96]    3:  ; in die MS-DOS-Kommandoebene zurück
[DX4]    4:        org  1ØØh      ; Programm startet ab
[ZØ3]    5:                       ; 1ØØhex: für COM-Da-
[6D3]    6:                       : teien erforderlich
[347]    7: start  jmps anfang ; Sprung Programmanfang
[ØØØ]    8:
[7W2]    9:        ; -- Datenbereich --
[VC9]   1Ø: text   db  13,1Ø,"Ein Assemblerprogramm für
          den SAM",13,1Ø,"$"
[ØØØ]   11:
[DK3]   12:        ; -- Programmbereich --
[574]   13: anfang mov dx,offset(text)
[TD3]   14:                       ; Adresse des Strings
[ZM5]   15:        mov ah,Ø9h ; String per Interrupt
[B35]   16:        int 21h    ; 21hex, Funktion Ø9hex
[MØ4]   17:                   ; am Bildschirm ausgeben
[2Z5]   18:        mov ah,4ch ; Programm per Interrupt
[Z45]   19:        int 21h    ; 21hex, Funktion 4Chex
[8C1]   2Ø:                   ; beenden
[DB5]   21:        end  start ; Programmende und De-
[AA3]   22:                   ; finition des Start-
[3H1]   23:                   ; punktes
```

Listing 2. ...und speziell angepaßt an den Assembler SAM III (in dieser Ausgabe)

Auflösung:

dezimal 5 = binär 101
dezimal 36 = binär 100100
dezimal 158 = binär 10011110
binär 01001001 = dezimal 73
binär 10101010 = dezimal 170
binär 10110111 = dezimal 183
dezimal 255 = hexadezimal FF
dezimal 616 = hexadezimal 268
dezimal 7654 = hexadezimal 1DE6

VirusDoktor

Abwehr von und Schutz vor Computerviren

Programmsammlung für MS-DOS-Computer

Bisher war der Anwender beim Thema "Computerviren" auf sich allein gestellt. Mit VirusDoktor sind Sie nun gegen alle bekannten PC-Computerviren gewappnet:

Fünf einzelne Programme leisten mehr als jede Komplettlösung.

SperrTSR
– Residentes Programm zum Sperren sämtlicher Schreibzugriffe auf Massenspeicher

SperrHD
– Software-Schreibschutz für Festplatten und Laufwerke

SperrINT
– Sperren des gesetzten Read-Only-Attributes über den Interrupt 21h

CMOS-BKP
– Backup-Programm für das C-MOS-RAM von AT- und AT-kompatiblen Rechnern

VirCheck
– Beobachtung und Kontrolle aller ausführbaren Dateien auf Massenspeicher

Für jedes Problem mit PC-Viren haben Sie mit dem VirusDoktor die richtige Antwort parat. Das Handbuch informiert Sie in einem ausführlichen Grundlagenteil über Systematik, Auftreten und Bekämpfung von PC-Computerviren.

Für MS-DOS ab 2.1 bis 3.3

99,– DM (unverbindliche Preisempfehlung)
Wenn Sie über den DMV-Bestellservice bestellen, gilt folgendes:

Inland:		Ausland:	
Einzelpreis	99,– DM	Einzelpreis	99,– DM
zzgl. Versandkosten	4,– DM	zzgl. Versandkosten	6,– DM
Endpreis	**103,– DM**	**Endpreis**	**105,– DM**

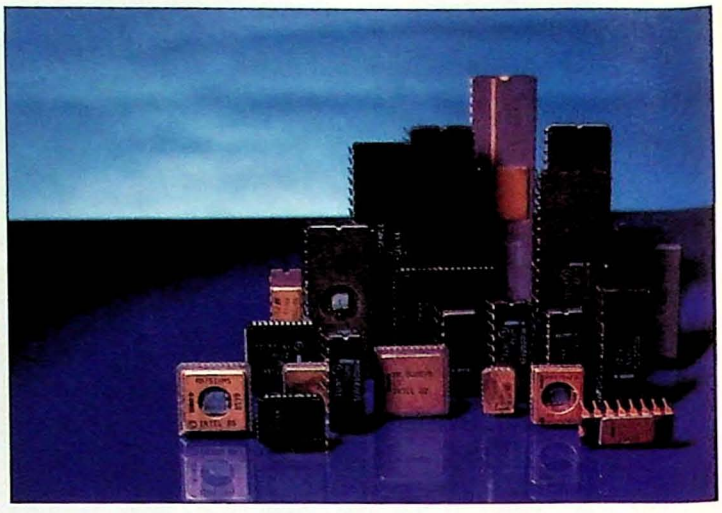
5 1/4"-Diskette Best.-Nr. 256 **3 1/2"-Diskette Best.-Nr. 257**

Bitte benutzen Sie die Bestellkarte.

DMV-Verlag · Postfach 250 · 3440 Eschwege

DMV Daten- und Medienverlag

Dateitypen und MS-DOS-Funktionen



Die Dateitypen »com« und »exe« bezeichnen ausführbare Programmdateien. Auch in Assembler können Sie solche Programme erzeugen. Nachdem Sie das dazu erforderliche Verfahren kennengelernt haben, geht es tiefer in die Geheimnisse der Assemblerprogrammierung hinein: Die MS-DOS- und BIOS-Funktionen anstelle eigener Routinen ersparen Entwicklungsenergie. Schließlich helfen Makros beim Programmieren – aber sind Unterprogramme nicht vorteilhafter?

Daß die Adressierung des Arbeitsspeichers etwas mit den Programmtypen »com« und »exe« zu tun haben könnte, ist nicht ohne weiteres einsichtig. Der 8086/8088-Prozessor verwaltet nämlich seinen Speicher über Segmentregister. Dies ist nötig, weil sich mit den 16-Bit-Registern des 8086/8088 nur 2^{16} = 65536 Speicherzellen (64 KByte) adressieren lassen, während der maximale Speicherbereich des Prozessors 1 MByte beträgt. Deshalb geben die Segmentregister des 8086/8088 jeweils die Startadresse (Basis) für ein Speichersegment an, und die Adresse in einem der normalen 16-Bit-Register (Offset) zeigt auf eine einzelne Speicherzelle innerhalb dieses Segments. Weil ein 16-Bit-Register höchstens 64 KByte Speicherbereich adressieren kann, ist auch die Länge eines Speichersegments auf maximal 64 KByte begrenzt. Ganz konkret wird die effektive Adresse einer Speicherzelle berechnet, indem die Segmentadresse in einem Segmentregister mit 16 multipliziert und zum Wert der Adresse eines normalen 16-Bit-Registers addiert wird. Enthält beispielsweise das Segmentregister DS die Adresse A000hex und das Register DX die Adresse 4000hex, so beträgt die effektive Adresse dieser beiden Register

```
A000hex x 10hex + 4000hex = A4000hex
```

Mit dem DX-Register läßt sich damit der Speicherbereich von A0000hex (DX = 0000hex) bis AFFFFhex (DX = FFFFhex) adressieren.
Jedes Segmentregister des 8086/8088 hat seinen eigenen Zuständigkeitsbereich. Das Segmentregister CS (Code Segment«) ist für den Befehlszeiger IP zuständig, SS (Stack Segment) gibt die Segmentadresse für den Stapelzeiger SP und den Basiszeiger BP an, und ES (Extra Segment) bestimmt das Speichersegment für das Zielregister DI bei Zeichenkettenoperationen. In allen anderen Fällen ist DS (Data Segment) angesprochen, so daß dieses Segmentregister für die Daten in den Registern AX, BX, CX, DX, SI und DI gilt.
In Assemblerprogrammen müssen Sie sich um diese Feinheiten nicht kümmern, wenn Ihre Programme nicht länger als 64 KByte sind – und welches Assemblerprogramm ist schon länger? Dann können Sie nämlich das Speichermodell für Dateien mit der Endung »com« verwenden, das mit einem einzigen Speichersegment arbeitet.
In diesem Speichersegment sind Programmcode und Daten gemeinsam enthalten. Der Programmcode beginnt ab der Adresse 0100hex mit einem Sprungbefehl auf den logischen Programmstart, unmittelbar von den Daten des Programms gefolgt. Daran schließt sich der eigentliche Programmcode an, und für den Stapel (Stack) reserviert MS-DOS automatisch 100hex Byte am oberen Ende des Speichersegments.

Com-Dateien arbeiten mit einem 256 Byte langen Programmvorspann, der als PSP bezeichnet wird

Wenn nun die Daten und der Programmcode zusammen weniger als 64 KByte minus 100hex Byte belegen und die Stapelgröße den Betrag von 100hex Byte tatsächlich nicht überschreitet, funktioniert das Verfahren einwandfrei. Überschneiden sich dagegen die Bytes von Programm, Daten und Stapel, so führt dies in der Regel zum Absturz des Programms.
Der Speicherbereich von Adresse 0000hex bis 00FFhex innerhalb des Speichersegments einer Datei mit der Endung »com« wird für das Kennzeichen dieses Dateityps benötigt: das PSP (Program Segment Prefix). Das PSP enthält Daten, die für die Verwaltung der Com-Datei durch das Betriebssystem erforderlich sind. Dieser Programmvorspann wird beim Aufruf einer Com-Datei aus der MS-DOS-Kommandoebene von MS-DOS selbsttätig angelegt, so daß sich der Programmierer darum nicht weiter kümmern muß.
Das Programmgerüst für Dateien, die mit dem Speichermodell der Com-Dateien arbeiten, sieht immer gleich aus. Zuerst wird mit dem Pseudobefehl »assume« den Segmentregistern CS und DS ein gemeinsames Speichersegment zugewiesen.

Der Aufbau eines Assemblerprogramms für Com-Dateien erfolgt nach einem festen Schema

Wenn das Programm keine Zeichenkettenoperationen enthält, muß das Segmentregister ES nicht initialisiert werden. Eine Zuweisung für das Stapelsegmentregister SS ist sogar überflüssig, weil es automatisch initialisiert wird. Zu diesem Zweck reserviert das Betriebssystem MS-DOS am oberen Ende des Speichersegments 100hex Byte und legt als Kennzeichen für die Stapelspitze (»top of stack«) in den Speicherzellen FFFEhex und FFFFhex die 16-Bit-Adresse 0000hex ab. Diese Adresse weist auf die erste Adresse im PSP, die wiederum die Rücksprungadresse zum Beenden des Programms enthält.
Weil das Speichermodell für Com-Dateien mit nur einem Speichersegment arbeitet, muß allen zwei oder drei Segmentregistern das gleiche Segment zugewiesen werden. Das Segment trägt in der Regel die Bezeichnung »code«, so daß der komplette Assume-Befehl

```
assume cs:code,ds:code
```

oder

```
assume cs:code,ds:code,es:code
```

lautet.

Der Linker »link« des MASM (Makroassembler) warnt in diesem Fall zwar mit »Warning: no stack segment«,daß kein Segment für den Stapel definiert wurde, doch diese Fehlermeldung können Sie getrost ignorieren, wenn Sie vorhaben, die vom Linker erzeugte Exe-Datei mit dem MS-DOS-Befehl »exe2bin« in eine Com-Datei umzuwandeln. Der nächste Befehl im Programm muß mit

```
code segment
```

den Start des Speichersegments definieren. In diesem Beispiel weist er gleichzeitig dem Segment den Namen »code« zu.
Der sich anschließende Pseudobefehl

```
org 0100h
```

gibt dem Assembler die Anweisung, das Programm ab der Adresse 0100hex im Speichersegment abzulegen, so daß die ersten 100hex Byte für das PSP reserviert bleiben. Nun muß der unbedingte Sprungbefehl »jmp« folgen, der mit der Marke, die im Befehl »end« steht, versehen ist und den eigentlichen Programmstart aufruft

```
marke: jmp anfang
```

Ein Programm, das mit dem Speichermodell der Com-Dateien arbeitet, darf nämlich nicht gleich »loslegen«, weil zu Programmbeginn der Datenbereich stehen muß, das heißt, der Speicherbereich, in dem Variablen, Konstanten und Datenpuffer definiert

Befehl	Funktion
AAA	Passe Ergebnis nach ASCII-Addition an
AAD	Passe Ergebnis nach ASCII-Division an
AAM	Passe Ergebnis nach ASCII-Multiplikation an
AAS	Passe Ergebnis nach ASCII-Subtraktion an
ADC	Addiere mit Übertrag
ADD	Addiere ohne Übertrag
AND	Führe logische AND-Verknüpfung aus
CALL	Rufe Unterprogramm auf
CBW	Wandle Byte in Wort um
CLC	Lösche Übertrag-Statusbit
CLD	Lösche Richtung-Statusbit
CLI	Lösche Unterbrechung-Statusbit
CMC	Invertiere Übertrag-Statusbit
CMP	Führe Vergleich aus
CMPS	Vergleiche Zeichenketten
CMPSB	Vergleiche Zeichenketten byteweise
CMPSW	Vergleiche Zeichenketten wortweise
CWD	Wandle Wort in Doppelwort um
DAA	Passe Ergebnis nach dezimaler Addition an
DAS	Passe Ergebnis nach dezimaler Subtraktion an
DEC	Um 1 vermindern
DIV	Führe vorzeichenlose Division aus
ESC	Übergib an externe Einheit
HLT	Halte Prozessor an
IDIV	Führe vorzeichenbehaftete Division aus
IMUL	Führe vorzeichenbehaftete Multiplikation aus
IN	Lies über Port ein
INC	Erhöhe um 1
INT	Unterbreche
INTO	Unterbreche bei Überlauf
IRET	Kehre von Unterbrechungsroutine zurück
JA	Verzweige, wenn darüber (entspricht JNBE)
JAE	Verzweige, wenn darüber oder gleich (entspricht JNB)
JB	Verzweige, wenn darunter (entspricht JNAE)
JBE	Verzweige, wenn darunter oder gleich (entspricht JNA)
JCXZ	Verzweige, wenn Inhalt von CX gleich 0
JE	Verzweige, wenn gleich (entspricht JZ)
JG	Verzweige, wenn größer (entspricht JNLE)
JGE	Verzweige, wenn größer oder gleich (entspricht JNL)
JL	Verzweige, wenn kleiner (entspricht JNGE)
JLE	Verzweige, wenn kleiner oder gleich (entspricht JNG)
JMP	Verzweige
JNA	Verzweige, wenn nicht darüber (entspricht JBE)
JNAE	Verzweige, wenn nicht darüber und nicht gleich (entspricht JB)
JNB	Verzweige, wenn nicht darunter (entspricht JAE)
JNBE	Verzweige, wenn nicht darunter und nicht gleich (entspricht JA)
JNE	Verzweige, wenn nicht gleich (entspricht JNZ)
JNG	Verzweige, wenn nicht größer (entspricht JLE)
JNGE	Verzweige, wenn nicht größer und nicht gleich (entspricht JL)
JNL	Verzweige, wenn nicht kleiner (entspricht JGE)
JNLE	Verzweige, wenn nicht kleiner und nicht gleich (entspricht JG)
JNO	Verzweige, wenn kein Überlauf
JNP	Verzweige, wenn keine Parität (entspricht JPO)
JNS	Verzweige, wenn positiv
JNZ	Verzweige, wenn nicht 0 (entspricht JNE)
JO	Verzweige, wenn Überlauf
JP	Verzweige, wenn Parität (entspricht JPE)
JPE	Verzweige, wenn Parität gerade (entspricht JP)
JPO	Verzweige, wenn Parität ungerade (entspricht JNP)
JS	Verzweige, wenn negativ
JZ	Verzweige, wenn 0 (entspricht JE)
LAHF	Lade untere 8 Statusbits in AH-Register
LDS	Lade DS-Register und weiteres Register
LEA	Berechne und lade effektive Adresse
LES	Lade ES-Register und weiteres Register
LOCK	Aktiviere Lock-Signal für nachfolgenden Befehl
LODS	Lade Zeichenkette
LODSB	Lade Zeichenkette byteweise
LODSW	Lade Zeichenkette wortweise
LOOP	Wiederhole Schleife bis Zähler CX gleich 0
LOOPE	Wiederhole Schleife bis ungleich oder Zähler CX gleich 0
LOOPNE	Wiederhole Schleife bis gleich oder Zähler CX gleich 0
LOOPNZ	Wiederhole Schleife bis gleich 0 oder Zähler CX gleich 0
LOOPZ	Wiederhole Schleife bis ungleich 0 oder Zähler CX gleich 0
MOV	Übertrage Zeichen
MOVS	Übertrage Zeichenkette
MOVESB	Übertrage Zeichenkette byteweise
MOVESW	Übertrage Zeichenkette wortweise
MUL	Führe vorzeichenlose Multiplikation aus
NEG	Negiere (2er-Komplement bilden)
NOP	Führe keine Operation aus
NOT	Invertiere (1er-Komplement bilden)
OR	Führe logische OR-Verknüpfung aus
OUT	Gib über Port aus
POP	Lies Wort vom Stack
POPF	Lies Statusbits vom Stack
PUSH	Lege Zeichen auf Stack
PUSHF	Lege Statusbits auf Stack
RCL	Rotiere durch Übertrag-Statusbit nach links
RCR	Rotiere durch Übertrag-Statusbit nach rechts
REP	Wiederhole Zeichenkettenoperation bis Zähler CX gleich 0
REPE	Wiederhole Zeichenkettenoperation bis ungleich oder Zähler CX gleich 0
REPNE	Wiederhole Zeichenkettenoperation bis gleich oder Zähler CX gleich 0
REPNZ	Wiederhole Zeichenkettenoperation bis gleich 0 oder Zähler CX gleich 0
REPZ	Wiederhole Zeichenkettenoperation bis ungleich 0 oder Zähler CX gleich 0
RET	Kehre von Unterprogramm zurück
ROL	Rotiere nach links
ROR	Rotiere nach rechts
SAHF	Lade AH-Register in untere 8 Statusbits
SAL	Schiebe arithmetisch nach links (entspricht SHL)
SAR	Schiebe arithmetisch nach rechts
SBB	Subtrahiere mit Übertrag
SCAS	Durchsuche Zeichenkette
SCASB	Durchsuche Zeichenkette byteweise
SCASW	Durchsuche Zeichenkette wortweise
SEG	Definiere gültiges Segmentregister für nachfolgenden Befehl
SHL	Schiebe logisch nach links (entspricht SAL)
SHR	Schiebe logisch nach rechts
STC	Setze Übertrag-Statusbit
STD	Setze Richtung-Statusbit
STI	Setze Unterbrechung-Statusbit
STOS	Speichere Zeichenkette
STOSB	Speichere Zeichenkette byteweise
STOSW	Speichere Zeichenkette wortweise
SUB	Subtrahiere ohne Übertrag
TEST	Vergleiche über AND-Verknüpfung
WAIT	Warte auf Testsignal
XCHG	Tausche Registerwerte (und Speicherinhalt) aus
XLAT	Adressiere Tabelle und lade Zeichen
XOR	Führe logische XOR-Verknüpfung aus

Tabelle. Die Befehle des 8086/8088-Prozessors geben Aufschluß über seine Leistungsfähigkeit

sind. Dieser Datenbereich muß sich an den Sprungbefehl unmittelbar anschließen. Danach erst beginnt das Programm, das mit dem Sprungbefehl gestartet wird.
Um das Programm elegant zu beenden, empfiehlt sich die Funktion 4Chex des MS-DOS-Interrupt 21hex, die eine Rückkehr in die MS-DOS-Kommandoebene bewirkt. Zwar schließt auch der MS-DOS-Interrupt 20hex ein Programm ab, doch ist dieser Interrupt veraltet, so daß er bei Dateien mit der Endung »exe« nicht korrekt arbeitet und aus Gründen der Aufwärtskompatibilität nicht benutzt werden sollte.

```
mov ah,4Ch
int 21h
```

lautet die korrekte Befehlsfolge zum Beenden des Programms. Zu guter Letzt wird mit den beiden Pseudobefehlen

```
code ends
```

und

```
end start
```

das Segmentende von »code« sowie das Programmende festgelegt. Die Angabe von »start« im Befehl »end« markiert die Zeile, in der die Sprungmarke »start« steht, als Einstiegspunkt beim Programmstart. Listing 1 zeigt zusammenfassend den beschriebenen Aufbau eines Assemblerprogramms nach dem Speichermodell der Com-Dateien.
Wenn Sie Ihre Assemblerprogramme mit diesem Grundgerüst aufbauen, das heißt, anstelle der Kommentare und Punkte Daten und Befehle einfügen, können Sie am Ende die erzeugte Com-Datei problemlos aus der MS-DOS-Kommandoebene aufrufen.

Das Speichermodell für Exe-Programme kann beliebig viele Segmente enthalten

Natürlich existiert für Dateien mit der Endung »exe« ein eigenes Speichermodell. Beim Speichermodell der Exe-Dateien können Sie im Unterschied zum Speichermodell für Com-Dateien beliebig viele Speichersegmente definieren. Listing 2 zeigt ein Beispiel, das mit einem Programmsegment, zwei Datensegmenten sowie einem Extrasegment und einem Stapelsegment arbeitet. Der Wechsel zwischen den Datensegmenten und die Initialisierung von zwei Stapelzeigern (System- und Programmstapelzeiger) geht aus diesem Beispiel ebenfalls deutlich hervor.
Interessant ist, daß bei diesem Speichermodell die Segmentregister DS, ES und SS einzeln initialisiert werden müssen. Während dies für das Segmentregister CS beim Programmstart automatisch geschieht (CS gibt schließlich mit dem Befehlszeiger IP die Startadresse des Programms an), muß die Initialisierung der übrigen Segmentregister durch den Programmierer erfolgen.
Zuerst wird jeweils das AX-Register mit der Segmentadresse des entsprechenden Speichersegments geladen und anschließend dieser Wert in das DS geschrieben. Ein direkter Befehl in der Art

```
mov ds,segment
```

ist beim 8086/8088-Prozessor leider nicht möglich, so daß der Umweg über das AX-Register notwendig ist.
Wie Sie sehen, ist beim Speichermodell für Exe-Dateien der Umgang mit den Segmenten und Segmentregistern schon etwas komplizierter, so daß sich zumindest für den Anfang das Speichermodell der Com-Dateien empfiehlt.

»exe2bin« wandelt unter bestimmten Voraussetzungen Exe- in Com-Dateien um

Wenn Sie mit dem MASM-Assembler arbeiten, vergessen Sie nicht, die vom Linker erzeugten Exe-Dateien mit dem MS-DOS-Befehl »exe2bin« in Com-Dateien umzuwandeln. Exe-Dateien, die nach dem Speichermodell der Com-Dateien aufgebaut sind, funktionieren nämlich nicht korrekt. Umgekehrt lassen sich Exe-Dateien, die mit mehreren Speichersegmenten arbeiten, von vornherein nicht in Com-Dateien konvertieren.
Listing 2 zeigt neben dem Aufbau des Speichermodells für Exe-Dateien auch den Pseudobefehl »dw« (define word). Dieser Befehl ähnelt dem Befehl »db«, nur definiert er nicht ein oder mehrere Bytes, sondern ein Wort beziehungsweise mehrere Worte. Während ein Byte als 8-Bit-Wert nur eine Speicherzelle belegt, ist das Wort ein 16-Bit-Wert und nimmt demzufolge zwei unmittelbar aufeinanderfolgende Speicherzellen ein. Dabei wird das niederwertige vor dem höherwertigen Byte im Speicher abgelegt. Dieses Verfahren ist als Intel-Format bekannt.
Listing 2 enthält auch eine neue Variante der Befehle »db« und »dw«. Hier bekommen die vor den Befehlen stehenden Variablen keine absoluten Werte zugewiesen, sondern es werden nur Speicherbereiche initialisiert. Diese Initialisierung geschieht über den Befehl »dup«. So initialisiert beispielsweise der Befehl

```
puffer db 100h dup(?)
```

einen Speicherbereich von 100hex Speicherzellen für die Variable »puffer«. Das Fragezeichen gibt in diesem Fall an, daß der Inhalt von »puffer« bei der Initialisierung gleichgültig ist. Soll dagegen nach der Initialisierung ein bestimmter Wert in »puffer« stehen, so muß dieser Wert anstelle des Fragezeichens in die Klammern gesetzt werden. So setzt beispielsweise der Befehl

```
puffer db 100h dup(0)
```

die 100hex Speicherzellen für »puffer« auf den Wert 0.

Vor der Erläuterung der Bedeutung der MS-DOS-Funktionen ist es angebracht, die wichtigsten Assemblerbefehle etwas genauer kennenzulernen, weil Ihnen sonst die Bedeutung der Programmbeispiele verschlossen bleibt.

Die Tabelle zeigt den gesamten Befehlssatz des 8086/8088-Prozessors in alphabetischer Reihenfolge. Von diesen Befehlen ist der Befehl »mov« zweifellos der wichtigste. Mit »mov« (für englisch »move«, bewegen) lassen sich Register und Speicherzellen auslesen und laden. Kein Programm kommt ohne den Mov-Befehl aus.
Der Befehl

```
mov bx,1
```

setzt beispielsweise das BX-Register auf 1. Der Befehl

```
mov flag,cl
```

schreibt in die durch den Wert der Marke »flag« bestimmte Speicherzelle den Wert des CL-Registers. Der Befehl

```
mov dx,offset text
```

lädt schließlich die Adresse der Variablen »text« in das DX-Register. Ohne die Angabe »offset« würde der Prozessor den Inhalt der Variablen »text« in das DX-Register schreiben.

Selten kommt es vor, daß ein Programm ausschließlich linear verläuft. Immer wieder sind im Programmfluß Verzweigungen notwendig. Hierfür existieren beim 8086/8088 die Sprungbefehle. Der Befehl »jmp« (für englisch »jump«, springen), den Sie bereits aus den Programmbeispielen kennen, ist ein unbedingter Sprungbefehl und entspricht dem Goto-Befehl in Basic-Programmen. Dem Befehl »jmp« muß eine Sprungmarke (ein Label) folgen, damit der Assembler weiß, auf welche Adresse sich dieser Befehl bezieht. Die Zieladresse des Sprungbefehls im Assemblerprogramm muß in der ersten Spalte mit dieser Sprungmarke und einem angehängten Doppelpunkt versehen sein. Mit

```
        jmp weiter
text    db "Dummy-Text"
weiter: mov ax,offset text
```

überspringt das Programm die Definition der Variablen »text« und fährt mit dem Mov-Befehl fort.
Die zweite Sorte der Sprungbefehle sind die bedingten Sprungbefehle. Diese Befehle verzweigen nur unter bestimmten Bedingungen und sind mit dem Basic-Befehl »if« vergleichbar. Die Angabe des Sprungziels erfolgt bei den bedingten Sprungbefehlen wie beim Befehl »jmp« auch über Sprungmarken.
Damit die Sprungbefehle einen Grund zum Verzweigen haben, muß zuvor ein Vergleich stattfinden, der die Statusbits (Flags), die einen bestimmten Zustand des Prozessors anzeigen, entsprechend setzt. Mögliche Zustände, die durch ein Statusbit gemeldet werden, sind:
- Übertrag ist aufgetreten/nicht aufgetreten (Carry Flag und Auxiliary Carry Flag)
- Parität ist gerade/ungerade (Parity Flag)
- Ergebnis lautet Null/nicht Null (Zero Flag)
- Vorzeichen ist negativ/positiv (Sign Flag)
- Einzelschrittmodus ist aktiv/nicht aktiv (Trace Flag)
- Unterbrechung ist aufgetreten/nicht aufgetreten (Interrupt Flag)
- Richtung ist vorwärts/rückwärts (Direction Flag)
- Überlauf ist aufgetreten/nicht aufgetreten (Overflow Flag)

Für Vergleiche, welche die Statusbits beeinflussen, ist der Befehl »cmp« (für englisch »compare«, vergleichen) vorgesehen, der zwei Register oder ein Register und einen Operanden miteinander vergleicht. Die Befehlsfolge

```
cmp bl,0
jz fehler
div bl
```

vergleicht beispielsweise den Inhalt des BL-Registers mit Null: Ist der Inhalt 0, so wird die Routine »fehler« aufgerufen, im anderen Fall erfolgt die Division von AX durch BL.
Für Unterprogrammaufrufe stehen die Befehle »call« und »ret« zur Verfügung. »call« (für englisch »call subroutine«) ruft ein Unterprogramm über eine Sprungmarke auf, und mit »ret« (für englisch »return from subroutine«) kehren Sie aus dem Unterprogramm in das Hauptprogramm zurück. Der Befehl »call« legt beim Aufruf eines Unterprogramms eine Rücksprungadresse auf dem Stapel ab, damit der Befehl »ret« im Unterprogramm diese Adresse als Zieladresse für die Rückkehr in das Hauptprogramm verwenden kann. Mit

```
: : :
call sub
: : :
end

sub: xor ax,ax
     xor bx,bx
     xor cx,cx
     xor dx,dx
     ret
```

setzt das Unterprogramm »sub« durch eine Exklusiv-Oder-Verknüpfung die Register AX, BX, CX und DX auf 0 zurück.
(Probieren Sie es aus, die Verknüpfung funktioniert immer und ist schneller als Befehle in der Art von »mov ax,0«.)

Mit den Befehlen »call« und »ret« lassen sich Unterprogramme aufrufen und beenden

Nunmehr besitzen Sie das Rüstzeug, das Sie zum Verständnis der Arbeitsweise von MS-DOS-Funktionen benötigen. Alle MS-DOS-Funktionen (und auch die BIOS-Funktionen) lassen sich über programmierbare Unterbrechungen, die als »Software-Interrupts« bezeichnet werden, aufrufen. Der 8086/8088-Prozessor kann maximal 256 verschiedene Software-Interrupts mit den Nummern 00 bis FFhex erzeugen. Für die MS-DOS-Funktionen ist der Software-Interrupt 21hex zuständig. Der Assemblerbefehl zum Aufruf dieses Interrupts lautet

```
int 21h
```

Wie Sie noch sehen werden, spielt der Int-Befehl beim PC eine äußerst wichtige Rolle, denn der Programmierer kann sich durch den Aufruf von MS-DOS- und BIOS-Funktionen über einen Software-Interrupt viel Arbeit sparen, weil diese Funktionen direkt auf die Routinen des Betriebssystems und des BIOS (Basic Input Output System) zugreifen und deren Fähigkeiten nutzen.
Durch das Konzept der Software-Interrupts müssen diese Routinen nicht über absolute Adressen angesprochen werden (die sich dann je nach MS-DOS- oder BIOS-Version ändern würden), sondern lassen sich über Software-Interrupts mit festen Nummern aufrufen.
Mit dem Befehl »int« wird also ein Wert im Bereich von 00 bis FFhex übergeben, der die Nummer des gewünschten Software-Interrupts bezeichnet. Zu jedem dieser Interrupts befindet sich im ersten KByte des PC-Speichers eine vier Byte lange Adresse. Der Prozessor sucht sich selbständig die zu dem Interrupt passende Adresse (auch »Interrupt-Vektor« genannt) und ruft diese Adresse, und damit die zu dem Interrupt gehörende Serviceroutine auf. Wenn der Software-Interrupt beendet ist, kehrt der Prozessor automatisch zu dem Befehl im aufrufenden Programm zurück, der dem Int-Befehl folgt.
Die meisten Software-Interrupts sind für einen bestimmten Bereich (Bildschirm, Datenträger, Tastatur und so weiter) zuständig. Weil nun zu jedem dieser Bereiche mehrere Betriebssystem- und BIOS-Routinen existieren, der Anwender jedoch immer nur eine bestimmte Routine oder Routinengruppe aufrufen möchte, sind diese einfach von Null beginnend durchnumeriert. Jede Routine oder Routinengruppe, der eine eigene Nummer zugewiesen ist, wird als Funktion bezeichnet. Um nun eine bestimmte Funktion aufzurufen, muß vor dem Int-Befehl die Nummer der gewünschten Funktion in das AH-Register geschrieben werden.
Damit es auch möglich ist, die einzelnen Routinen einer Routinengruppe anzusprechen, sind diese oft ebenfalls von Null beginnend durchnumeriert. Um die Routine einer Routinengruppe, auch Unterfunktion genannt, aufzurufen, müssen Sie vor dem Int-Befehl die Nummer der Funktion in das AH-Register und zusätzlich die Nummer der Unterfunktion in das AL-Register schreiben.
Benötigt eine Funktion oder Unterfunktion die Eingabe von Parametern oder gibt sie Parameter zurück (eines von beiden ist fast immer der Fall), so sind für diese Parameter bestimmte Register speziell reserviert. In der DOS Extra, Ausgabe 5, sind auf elf Seiten die Software-Interrupts des PC mit allen Funktionen und Unterfunktionen sowie den für die Parameter reservierten Registern aufgeführt. Auch Aufgabe und Bedeutung der Parameter werden erklärt.

Betriebssystem- und BIOS-Funktionen anstelle eigener Routinen sparen Entwicklungszeit

Ein typisches Beispiel für die Anwendung der Betriebssystem- und BIOS-Routinen ist der Aufruf der Funktion 00hex des Video-Interrupt 10hex. Diese Funktion stellt den Videomodus (Text- oder Grafikmodus) des verwendeten Bildschirmadapters ein. So ruft beispielsweise die Befehlsfolge

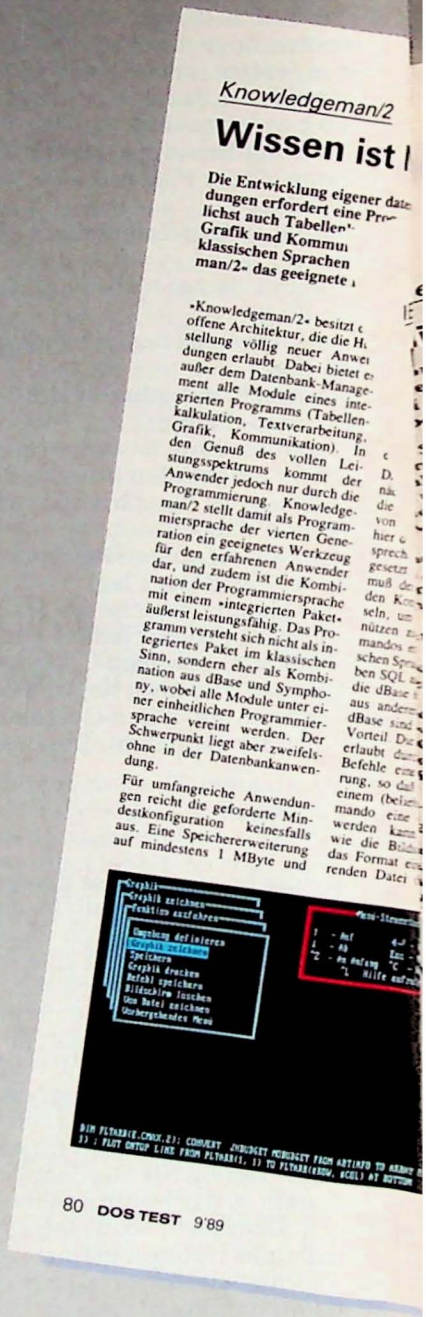
```
mov al,07h
mov ah,00h
int 10h
```

den Videomodus 7 des Monochromadapters (MDA) beziehungsweise der Hercules-Grafikkarte auf. In diesem Beispiel ist der Videomodus ein Ein-

DIE LOGISCHE

Das DOS-TEST-Magazin versorgt Sie jeden Monat mit aktuellen Hard- und Softwaretests, Vorberichten und Meldungen über sämtliche Entwicklungen auf dem Computermarkt.

Deshalb ein völlig neuartiges Konzept



DMV-Verlag · Postfach 250 · 3440 Eschwege

ERGÄNZUNG

NEU

Das neue Test-Schema:

Checkliste der wichtigsten Produktdaten, Leistungsumfang auf einen Blick.
Labortest mit präzisen Meßwerten von ausgewählten, anwenderbezogenen Programmfunktionen. Direkter Vergleich mit dem Referenzprodukt.
Praxistest der Leistungen im täglichen Einsatz.
Fazit mit Zielgruppenbestimmung und Anwenderempfehlung
Gesamturteil mit objektiver Testnote

Textmaker 2.01

Spitzenleistung mit Schönheitsfehlern

Textverarbeitungsprogramme gibt es in Preisklassen zwischen 50 und 2000 Mark. Die Unterschiede in der Leistung sind nicht immer so groß. Der »Textmaker« aus der kleinen »Softmaker-Schmiede« leistet für 248 Mark Beachtliches.

Einen guten Eindruck macht der »Textmaker 2.01« bei der Installation des nicht kopiergeschützten Programms. Mit Hilfe einer leicht verständlichen Dokumentation kann auch der Laie das Programm auf seiner Arbeitsdiskette oder Festplatte installieren und bedienen. Der Einsatz einer Festplatte ist nicht zuletzt aus Geschwindigkeitsgründen ratsam. Zum Leistungsumfang des Textmaker gehören, neben einem umfangreichen Rechtschreiblexikon und einer Silbentrennung, eine Adreßverwaltung, Grafikeinbindung und Liniengrafik.

Weitere Funktionen zum automatischen Anlegen eines Inhalts- und Indexverzeichnisses sowie die Fußnotenverwaltung machen den Textmaker auch für wissenschaftliche Arbeiten tauglich und beschränken ihn nicht auf das bloße Briefschreiben. Die Fußnoten werden dabei automatisch durchnumeriert und am Fußende der Seite, in der das zugehörige Zitat steht, ausgegeben. Die Handhabung dieser drei Funktionen ist erstaunlich unkompliziert.

Die Benutzeroberfläche besteht aus einer Menüleiste mit Pulldown-Menüs, die sowohl über die Tastatur als auch mit der Maus zu steuern sind. Bei der Textbearbeitung können Sie zum Beispiel für Seitenformatierungen auch Wordstar-kompatible Tastenbefehle (Punktbefehle) verwenden, wie aber auch alle anderen Funktionen von den Blockoperationen bis hin zu den Layoutvorgaben. Die jeweiligen Pulldown-Menüs beinhalten übersichtlicherweise und ähnlich einer Hilfefunktion zu jedem Befehl den zugehörigen Control-Code. Für Routiniers stellt besonders bei häufig wiederkehrenden Befehlen die Variante über die Control-Tastenkombinationen oft die schnellere Lösung dar.

Die Textbearbeitung erfolgt wahlweise im Text- oder Grafikmodus. Im Grafikmodus sehen Sie fast alle Textformatierungen so, wie sie zu Papier gebracht werden (WYSIWYG-Prinzip). Zwar kann das Programm auch Proportionalschrift im Blocksatz auf dem Drucker ausgeben, allerdings gilt dies nicht für die Bildschirmausgabe. Ein weiterer Vorteil des Grafikmodus ist das sogenannte »Page Preview«, anhand dessen Sie die Layout-Gestaltung von bis zu 20 Seiten vor dem Ausdruck auf dem Monitor kontrollieren können. Um Ihren Text ansehnlich zu gestalten, bietet Ihnen der Textmaker acht verschiedene Schrifttypen und Hervorhebungen wie Fettdruck, Unterstreichung oder Kursivschrift an. Des weiteren sorgen zentrierte Überschriften und optionaler Block- oder Flattersatz für ein professionelles Äußeres.

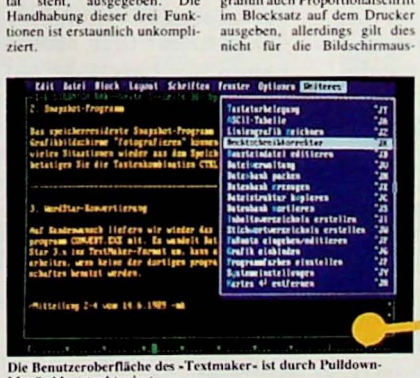
In neun Fenstern lassen sich mehrere Textsegmente gleichzeitig und unabhängig voneinander bearbeiten. Um Ihre Daten nicht zu verlieren, können Sie den Textmaker zur automatischen Datensicherung in Zeitintervallen von einer Minute bis zu einem ganzen Tag voreinstellen.

Des weiteren schreibt und liest der Textmaker ASCII-Dateien, so daß leicht Daten mit anderen Programmen ausgetauscht werden können. Textmakers Silbentrennung zeichnet sich durch eine hohe Treffsicherheit aus. Ihre Rechtschreibung lassen Sie mit dem 120000 Vokabeln starken Wörterbuch überprüfen, das Sie selbst erweitern können.

Lobenswerte Eigenschaften des Programms sind darüber hinaus die automatische Korrekturfunktion, die sofort jedes falsch geschriebene Wort verbessert, und die Fähigkeit, nach ähnlich klingenden Wörtern zu suchen (phonetische Suche). Außerdem können Sie Tastenkürzel vergeben, mit denen Sie Floskeln wie »Mit freundlichen Grüßen« auf »mfg« abkürzen und mit einem Tastendruck einfügen können.

Grafiken lassen sich problemlos in Texte einbinden. Zum einen können Sie mit dem zum Textmaker gehörenden Snapshot-Utility selbst Grafikdateien im PCX-Format generieren, zum anderen auch bereits vorhandene PCX-Dateien importieren. Da der Textmaker keine Farbdrucker unterstützt, empfiehlt es sich jedoch, nur Schwarzweiß-Grafiken zu verwenden. Ansonsten ist für die Druckausgabe hinreichend gesorgt: Die mitgelieferten 230 Druckertreiber unterstützen vom 9-Nadel-Drucker bis zum Laserdrucker alle gängigen Modelle. Sollte dies noch nicht genügen, wird auf Wunsch der noch nicht unterstützte Drucker kostenlos angepaßt. Der fehlende Treiber wird programmiert, und im Laufe der Zeit wird somit jedes Gerät erfaßt.

Nicht nur für Einsteiger stellt Textmaker ein interessantes Textverarbeitungsprogramm dar. Aufgrund des großen Funktionsumfangs ist das leicht zu bedienende Programm trotz kleiner Schwächen allemal seinen Preis wert. (as)



Die Benutzeroberfläche des »Textmaker« ist durch Pulldown-Menüs klar strukturiert

Textmaker 2.01

Preis: 248 Mark
Funktion: Textverarbeitung
Voraussetzungen: 256 KByte Arbeitsspeicher, MS-DOS ab Version 2.0, ein Diskettenlaufwerk, für Rechtschreibprüfung 512 KByte und Festplatte empfohlen
Info: Softmaker GmbH, Grabbestr. 9, 8500 Nürnberg 90, Tel.: 0911/303333

Checkliste

- ● Mausunterstützung
- ○ EMS-Unterstützung
- ● Undo-Funktion
- ● Textgröße über 64 KByte
- ● Suchen/Ersetzen
- ● ASCII lesen/schreiben
- ○ Makroprogrammierung
- ● Silbentrennung
- ● Rechtschreibprüfung
- ● Serienbrieffunktion
- ● Grafikmodus
- ● Page Preview
- ● Spaltensatz
- ● Grafikeinbindung
- ● Kopf-/Fußzeilen
- ● Fußnotenverwaltung
- ● Textbausteinverwaltung
- ● Fensterfunktionen
- ○ Druckformatvorlagen

Labortest

	Wert	Faktor[1]
Speichern	3,26 s	0,73
Suchen/Ersetzen	74,94 s	0,38
Serienbrief[2]	22,71 s	0,54

[1] Relativ zum Referenzprodukt Word 4.0
[2] Ausgabe auf Festplatte

Praxistest

Bedienung	gut
Funktionsumfang	gut
Ausgabefunktionen	gut
Datensicherheit	gut
Dokumentation	befriedigend

Fazit

Textmaker ist ein Textprogramm mit vielen, zum Teil originellen Funktionen. Sein Leistungsspektrum und der günstige Preis machen ihn für einen großen Anwenderkreis interessant. Nicht nur für Einsteiger empfehlenswert.

DOS TEST gut

In der **Überschrift** finden Sie auf den ersten Blick den Produktnamen, damit Sie nicht lange suchen müssen, was wir für Sie testen.

Im **Text** erhalten Sie umfassende Informationen zum Leistungs- und Lieferumfang sowie eine ausführliche Begründung der vergebenen Noten.

Der **Testkasten** enthält eine Zusammenfassung der wichtigsten Daten und Meßwerte. So können Sie sich schnell informieren und die Ergebnisse mit Tests anderer Produkte vergleichen.

Die **Produktinformation** nennt Ihnen Preis, Bezugsquelle und die für den Betrieb des Produkts notwendigen Hard- und Softwarevoraussetzungen.

In der **Checkliste** finden Sie die wichtigsten Produktdaten. Auf einen Blick bekommen Sie einen objektiven Überblick über den Leistungsumfang.

Der **Labortest** liefert Ihnen präzise Meßwerte von ausgewählten, anwenderbezogenen Programmfunktionen. Am Referenzfaktor lesen Sie ab, wie das Testprodukt im direkten Vergleich mit dem Referenzprodukt abschneidet.

Im **Praxistest** wird die Leistung im täglichen Einsatz benotet.

Das **Fazit** begründet das Gesamturteil und gibt eine Empfehlung, für welche Zielgruppe oder Anwendung das Produkt geeignet ist.

Das **Produktfoto** vermittelt Ihnen einen Eindruck vom optischen Erscheinungsbild und zeigt Ihnen wichtige Details.

Das **Gesamturteil** spiegelt die Leistungsfähigkeit im Vergleich zu den Mitbewerbern wider.

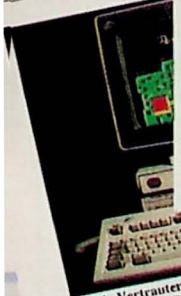


Bild 1. Vertrauter Anblick, revolutionäre Leistung: 486-Prozess... gesteu...

Am 28. August bei Ihrem Zeitschriftenhändler

DMV Daten- und Medienverlag

gabeparameter und wird im AL-Register übergeben.

Sie werden vielleicht bereits bemerkt haben, daß man das Beispiel auch kürzer schreiben kann, weil sich die Register AL und AH zum AX-Register zusammenfassen lassen:

```
mov ax,0007h
int 10h
```

Ein weiteres Beispiel für die Nutzung einer Betriebssystemroutine ist der Aufruf der MS-DOS-Funktion 02hex über den Software-Interrupt 21hex. Die MS-DOS-Funktion 02hex gibt ein einzelnes ASCII-Zeichen auf den Bildschirm (oder das aktuelle Ausgabegerät) aus. Der Code des ASCII-Zeichens muß sich als Eingabeparameter im DL-Register befinden. Diese Befehlsfolge gibt den Buchstaben »A« auf den Bildschirm aus:

```
mov dl,41h
mov ah,02h
int 21h
```

Die Funktionen der Software-Interrupts des PC reichen eigentlich zum Programmieren aller gängigen Routinen vollständig aus. So stellen die Software-Interrupts Funktionen zur Dateneingabe und Datenausgabe über Tastatur, Bildschirm und Drucker, Funktionen zur Datenträger- und Dateiverwaltung sowie Funktionen zur Programmverwaltung und Programmsteuerung zur Verfügung. Nur wenn Ihre Programme besonders schnell arbeiten sollen (beispielsweise bei der Bildschirmausgabe), ist es sinnvoll, sich für die MS-DOS- und BIOS-Funktionen eigene Routinen zu schreiben, die direkt auf Bildschirmspeicher, Tastatur oder Schnittstellen zugreifen. Das kostet dann allerdings Zeit und setzt fundierte Kenntnisse der Computer-Hardware und der Arbeitsweise von Betriebssystem und BIOS voraus.

Nachdem Sie in »Assembler zum ersten...« bereits erfahren haben, wie sich in Assemblersprache Text auf den Bildschirm ausgeben läßt, lernen Sie nun das Gegenteil kennen – wie Sie nämlich Text über die Tastatur einlesen können. Ihr nächstes Assemblerprogramm soll den Basic-Befehl

```
input "Bitte geben Sie Ihren Namen ein:
 ";name$
```

korrekt nachbilden, also zuerst den Text »Bitte geben Sie Ihren Namen ein: « ausgeben und darauf einen Text von der Tastatur lesen, der unter einer Variablen gespeichert wird.

Zur Textausgabe auf dem Bildschirm benutzen Sie in der Assemblersprache wieder den Software-Interrupt 21hex mit der MS-DOS-Funktion 09hex. Im DX-Register übergeben Sie die Startadresse der auszugebenden Zeichenkette, und das Ende der Zeichenkette markieren Sie mit einem Dollarzeichen. Im Programmbereich muß also stehen:

```
mov dx,offset text
mov ah,09h
int 21h
```

und im Datenbereich

```
text db "Bitte geben Sie Ihren Namen ein:
 $"
```

Für die Eingabe von Text eignet sich die Funktion 0Ahex des Interrupt 21hex. Diese Funktion läßt sich wie die Funktion 09hex aufrufen, nur enthält das DX-Register diesmal die Adresse eines Puffers, in den der eingelesene Text abgelegt wird. Die Funktion 0Ahex speichert alle über die Tastatur eingegebenen Zeichen ab der durch das DX-Register definierten Adresse. Das Einlesen von der Tastatur beenden Sie mit der Enter-Taste, oder Sie brechen mit [Ctrl-C] ab.

Ein großer Vorteil der Funktion 0Ahex besteht darin, daß alle eingegebenen Zeichen auch auf dem Bildschirm angezeigt werden und die Zeichenkette bis zum Drücken von [Enter] oder [Ctrl-C] frei editiert werden kann. Dies entlastet den Programmierer beim Schreiben einer Eingaberoutine deutlich.

Der Eingabepuffer für die Funktion 0Ahex muß folgendermaßen aufgebaut sein:

1. Byte: Anzahl der maximal folgenden Zeichen (legt die maximale Länge der Zeichenkette -1 fest)
2. Byte: Anzahl der tatsächlich eingegebenen Zeichen
3. Byte: erstes eingegebenes Zeichen
4. Byte: zweites eingegebenes Zeichen und so weiter.

Beginnt der Puffer beispielsweise mit dem Wert 21, so lassen sich maximal 20 Zeichen eingeben, weil das zweite Byte bereits für die Anzahl der tatsächlich eingegebenen Zeichen reserviert ist.

Listing 3 zeigt das komplette Assemblerprogramm »name«. Das Programm arbeitet wieder mit dem (einfacheren) Speichermodell für Com-Dateien.

Den aufmerksamen Lesern ist sicherlich nicht entgangen, daß die Befehlsfolge

```
mov dx,offset adresse
mov ah,nummer
int 21h
```

zum Aufruf einer Funktion des Interrupt 21hex zweimal im Programm »name« auftaucht (wenn auch mit unterschiedlichen Werten für die Zeichenkettenadresse »adresse« und die Funktionsnummer »nummer«). In längeren Programmen kommt diese Befehlsfolge noch weitaus öfter zum Einsatz, doch muß sie deshalb nicht jedesmal neu eingegeben werden.

Der MASM ist nämlich, wie schon sein voll ausgeschriebener Name »Makroassembler« verrät, in der Lage, sogenannte »Makros« zu verwalten. Makros sind immer wiederkehrende Befehlsfolgen, die der Programmierer bei der Arbeit mit einem Makroassembler als Makro definieren kann und dadurch nur ein einziges Mal schreiben muß. Wenn er die in einem Makro enthaltenen Befehle an einer bestimmten Stelle in seinem Programm benötigt, setzt er lediglich den Namen des Makros ein. Der Makroassembler fügt dann beim Übersetzen des Programms in Maschinencode den Inhalt des Makros automatisch an den entsprechenden Stellen ein.

Makros oder Unterprogramme: Makros sind eindeutig schneller

Anstelle eines Makros könnte man auch mit einem Unterprogramm arbeiten. Doch im Gegensatz zum Makro muß das Unterprogramm jedesmal zeitraubend über den Befehl »call« aufgerufen und mit dem Befehl »ret« beendet werden, während das Makro direkt an der gewünschten Stelle im Programm vorhanden ist und den Programmlauf nicht bremst.

Dafür hat ein Unterprogramm aber auch einen Vorteil: Es ist nämlich nur ein einziges Mal im Programm vorhanden, während der Assembler das Makro an jeder Stelle, an der es benötigt wird, neu einfügt. Bei langen Makros, die oft verwendet werden, kann dadurch ein Programm um ein ganzes Stück anwachsen.

Legen Sie also Wert auf Geschwindigkeit, sind Makros geeignet; wenn Sie dagegen Speicherplatz sparen wollen (oder müssen), greifen Sie besser auf Unterprogramme zurück. Das Makro für den oben angegebenen Aufruf einer Funktion des Interrupt 21hex lautet beim MASM:

```
funktion macro nummer,adresse
 mov dx,offset adresse
 mov ah,nummer
 int 21h
endm
```

Das Makro trägt damit den Namen »funktion« und erwartet die Übergabe der Parameter »nummer« und »adresse«. Der erste Parameter dient als Funktionsnummer für das AH-Register und der zweite Parameter als Speicheradresse für das DX-Register. Die beiden Pseudobefehle »macro« und »endm« kennzeichnen Anfang und Ende des Makros.

Wenn Sie dieses Makro zum Beginn des Programms »name« definieren,

können Sie für den Aufruf der Funktion 09hex in Listing 3

```
funktion 09h,text
```

und für den Aufruf der Funktion 0Ahex

```
funktion 0Ah,max
```

schreiben.
Der Assembler fügt dadurch beim Übersetzen des Programms in Maschinencode anstelle der beiden aufgeführten Zeilen jeweils das Makro »funktion« ein und ordnet im ersten Fall dem Parameter »nummer« den Wert 09hex und dem Parameter »adresse« die Variable »text« zu.

Im zweiten Fall wird »nummer« auf 0Ahex und »adresse« auf »max« gesetzt. Das bedeutet, daß die Angaben, die in einem Makroaufruf hinter dem Makronamen stehen, an die symbolischen Parameter in der Makrodefinition (in der gleichen Reihenfolge) übergeben werden.

Listing 4 zeigt, wie Sie das Programm »name« für den Einsatz des beschriebenen Makros abändern müssen.

(Markus Zietlow/Martin Althaus/al)

```
[7V6]    1: assume cs:code,ds:code ; Segmentzuweisun-
[L93]    2:                        ;gen für CS und DS
[RD6]    3: code   segment   ; Speichersegmentstart
[SK5]    4:        org  100h ; Programmstart ab 100hex
[000]    5:
[PP6]    6: start: jmp  anfang ; Sprung zum Programm
[Z91]    7:                     ;anfang
[163]    8:           ; -- Datenbereich    --
[DK3]    9:           ; -- Programmbereich --
[000]   10:
[836]   11: anfang: ...      ; Erster Programmbefehl
[YC0]   12:         ...
[VV4]   13:         ...      ; Letzter Programmbefehl
[M86]   14:       mov ah,4ch ; Programm über Interrupt
[9L5]   15:       int 21h    ; Funktion 4Chex beenden
[000]   16:
[1Y5]   17: code ends        ; Ende  Speichersegment
[N26]   18:       end  start ; Programmende und Defini-
[BW3]   19:                  ;tion des Startpunktes
```

Listing 1. Der grundlegende Aufbau von Assemblerprogrammen, die mit dem Speichermodell für Com-Dateien arbeiten

```
[8P2]    1:       title     exedatei
[SD4]    2:         ;    -- Segmentzuweisungen --
[367]    3: assume  cs:code,ds:data0,es:extra,ss:stack
[VF4]    4:         ;    -- Erstes Datensegment --
[YE2]    5: data0   segment
[CS2]    6: a       db        10   ;a=10
[KS2]    7: b       db        11   ;b=11
[SS2]    8: c       db        12   ;c=12
[LS1]    9: data0   ends
[4S4]   10:         ;     -- Zweites Datensegment --
[0F2]   11: data1   segment
[YS2]   12: d       db        13   ;d=13
[4T2]   13: e       db        14   ;e=14
[AT2]   14: f       db        15   ;f=15
[NS1]   15: data1   ends
[6Z2]   16:   ;     -Extrasegment
[SK2]   17: extra   segment
[5S5]   18: string1 db   100h dup (?) ;Platz für 3
[3W5]   19: string2 db   100h dup (?) ;max. 256 Bytes
[KZ5]   20: string3 db   100h dup (?) ;lange Strings
[DW1]   21: extra   ends
[AH3]   22:         ;     -- Stapelsegment --
[0E3]   23: stack   segment stack
[DV3]   24:         dw      400h dup (?) ;1 KByte
[HP5]   25: tos     label   word         ;für System-
[XL3]   26:                              ;stapel reservieren
[3H4]   27:         dw      400h dup (?) ;1 KByte für
[NW2]   28:                              ;Programmstapel
[XN5]   29: tob     label   word         ;reservieren
[PU1]   30: stack   ends
[2V2]   31:         ;   -- Codesegment --
[AD2]   32: code    segment
[PW6]   33: start:  mov  ax,data0 ;DS mit Datensegment
[PZ4]   34:         mov  ds,ax    ;data0 initialis.
[2H5]   35:         mov  ax,extra ;ES initialisieren
[GS1]   36:         mov  es,ax
[7G5]   37:         mov  ax,stack ;SS initialisieren
[UT1]   38:         mov  ss,ax
[LE5]   39:         mov  sp,offset tos ;SP initialis.
[BA5]   40:         mov  bp,offset tob ;BP initialis.
[LR3]   41:         ;    -- Weitere Befehle --
[S41]   42:         ;            ...
[PP5]   43:         mov  ah,4ch ;Programm über Inter-
[515]   44:         int  21h    ;rupt 21hex,Funktion
[VD2]   45:                     ;4Chex beenden.
[DM6]   46: code    ends        ;Ende v.Segment »code«
[TV4]   47:         end  start  ;Programmende und
[N25]   48:                     ;Definition des Startpunktes
```

Listing 2. Assemblerprogramme, die mit dem Speichermodell für Exe-Dateien arbeiten, sind im Aufbau komplexer als Com-Dateien

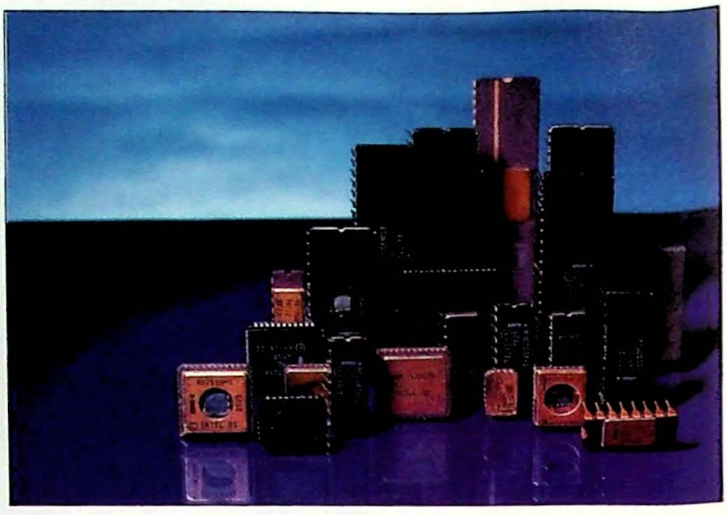
```
[WC6]    1:        title   name   ;Programmname »name«
[YL5]    2:   ; Dieses Programm gibt auf den Bild-
[7G6]    3:   ; schirm eine Zeichenkette aus und liest
[LX3]    4:   ;  eine Zeichenkette ein
[T97]    5: assume  cs:code,ds:code ;Segment »code«
[864]    6:                  ; ist für die Segmentregi-
[1G4]    7:                  ; ster CS und DS vorgesehen
[000]    8:
[L97]    9: code   segment   ; Segmentstart von »code«
[SK5]   10:        org  100h ; Programmstart ab 100hex
[MX3]   11:                  ; obligat bei COM-Dateien
[AV6]   12: start: jmp anfang ;Sprung nach »anfang«
[000]   13:
[FD3]   14:        ;    --   Datenbereich  --
[000]   15:
[8N7]   16: text   db  13,10,"Bitte geben Sie Ihren Nam
            en ein: $"
[FA5]   17: max    db  21       ; 1. Byte des Eingabe-
[DY3]   18:                     ;puffers (Pufferlänge)
[V94]   19:        db  21 dup (?) ; Eingabepuffer
[000]   20:
[EV3]   21:        ;    -- Programmbereich --
[000]   22:
[2S6]   23: anfang: mov  dx,offset text ; Adresse der
[M92]   24:                      ;Zeichenkette
[TV1]   25:         mov  ah,09h
[C85]   26:         int  21h     ;Zeichenkette über
[6U2]   27:                      ;Interrupt 21hex,
[WA3]   28:                      ;Funktion 09hex auf
[3X2]   29:                      ;Bildschirm geben
[6H5]   30:       mov dx,offset max  ;Pufferadresse
[VY5]   31:       mov ah,0ah     ; Zeichenkette einlesen
[381]   32:       int 21h
[GD5]   33:       mov ah,4ch     ; Programmende über
[6N5]   34:       int 21h        ; Interrupt 21hex Funk-
[NV1]   35:                      ; tion 4Chex
[W56]   36: code ends         ; Ende von Segment »code«
[BZ5]   37:       end start   ; Programmende und Defini-
[XZ3]   38:                   ; tion des Startpunktes
```

Listing 3. Das Assemblerprogramm »name« gibt eine Zeichenkette aus und liest eine Zeichenkette ein

```
[MD2]    1:       title     m_name
[Z46]    2:  ; Dieses Programm gibt über ein Makro
[8C6]    3:  ; eine Zeichenkette auf dem Bildschirm
[436]    4:  ; aus und liest eine Zeichenkette ein
[5L5]    5: funktion macro funktion,adresse
[7W2]    6:                        ;Makrodefinition
[X04]    7:         mov   dx,offset adresse
[SU2]    8:         mov   ah,funktion
[N81]    9:         int   21h
[HP0]   10:         endm
[0S6]   11: assume cs:code,ds:code ;Segment »code«
[5F3]   12:              ; ist für die Segment-
[EF4]   13:              ; register CS, DS vorgesehen
[KD7]   14: code   segment   ;Start von Segment »code«
[2X4]   15:         org  100h ;Programm startet ab
[5T3]   16:              ; 100hex (für COM-Dateien
[WH2]   17:              ; erforderlich)
[2U4]   18: start: jmp  anfang ;Sprung zum
[5S2]   19:                    ; Programmanfang
[183]   20:         ;       -- Datenbereich --
[AL7]   21: text   db   13,10,"Bitte geben Sie Ihren Na
            men ein: $"
[T45]   22: max    db   21     ;1. Byte des Eingabe-
[DY3]   23:                    ;puffers (Pufferlänge)
[KH4]   24:         db   21 dup (?) ;Eingabepuffer
[4W3]   25:         ;       -- Programmbereich --
[7A4]   26: anfang: funktion  09h,text
[KT3]   27:                    ;Zeichenkette ausgeben
[NR2]   28:         funktion  0ah,max
[CU3]   29:                    ;Zeichenkette einlesen
[PP5]   30:         mov  ah,4ch ;Programm über Inter-
[ZX4]   31:         int  21h    ;rupt 21hex, Funktion
[382]   32:                     ;4Chex beenden
[ZB6]   33: code    ends        ;Ende v.Segment »code«
[DP5]   34:         end  start ;Programmende und Defi-
[XC4]   35:                    ;nition des Startpunktes
```

Listing 4. Das Assemblerprogramm »m_name« ist die Makroversion von »name«

Die Pseudobefehle des MASM 4.0 und SAM III



Nachdem Sie bereits erste Erfahrungen im Umgang mit der Assemblersprache gesammelt haben und Ihnen mit SAM III auch ein leistungsfähiger Assembler für eigene Experimente zur Verfügung steht, finden Sie in diesem Kapitel Informationen zu den Pseudobefehlen des MASM und deren Programmierung.

Ein Assemblerprogramm setzt sich aus einer Abfolge von Assembleranweisungen zusammen, die eine bestimmte Aufgabe ausführen sollen. Eine Assembleranweisung besteht dabei entweder aus einem Assemblerbefehl oder einem sogenannten Pseudobefehl. Bekanntlich sind die Assemblerbefehle eine symbolische Darstellung der Maschinensprachebefehle des Prozessors. Im Gegensatz dazu weisen die Pseudobefehle den Assembler an, wie er die bereitgestellten Daten verarbeiten und ausgeben soll. Das Präfix »Pseudo« (griechisch für »Schein«) wurde gewählt, weil es sich bei den Befehlen dieser Gruppe nicht um echte Prozessorbefehle, sondern um von den Assemblerprogrammierern entwickelte Befehle handelt.

Pseudobefehle werden hauptsächlich dazu benutzt, Symbole zu definieren, Speicherbereiche zuzuweisen oder andere Arbeiten durchzuführen, die die Programmkonzeption und Programmumsetzung erleichtern. Im Gegensatz zu Assemblerbefehlen erzeugen die Pseudobefehle jedoch keinen Maschinencode.

Eine Anweisung, die einen Pseudobefehl enthält, besteht wie ein Assemblerbefehl aus einem Marken-, einem Befehls-, einem Operanden- und einem Kommentarfeld. Bis auf das Befehlsfeld sind alle Felder optional und müssen nicht belegt sein.

Genau wie bei Assemblerbefehlen können sich die Felder der Pseudobefehle an jeder Position innerhalb einer Zeile befinden. Sie müssen jedoch mindestens durch ein Leerzeichen oder einen Tabulator voneinander getrennt sein.

Der MASM 4.0 von Microsoft verfügt über mehr als 50 verschiedene Pseudobefehle, SAM III enthält eine Teilmenge davon. Die Tabelle zeigt die Pseudobefehle der beiden Assembler und deren unterschiedliche Form im Überblick.

Mit den Pseudobefehlen für die Symboldefinition sind Sie in der Lage, mathematischen Ausdrücken oder absoluten Zahlen symbolische Namen zuzuweisen. Die zu diesem Zweck benutzten Befehle »equ« und » = « ähneln sich sehr. Der Unterschied liegt darin, daß die mit dem Gleichheitszeichen festgelegten Symbole umdefiniert werden können. Zulässige Formen für die Symboldefinition sind beispielsweise:

```
akku equ  ax
index equ [bp+2]
kbyte=1024
```

Die am häufigsten benötigten Pseudobefehle sind die der Datendefinition. Fast alle Assemblerprogramme verwenden bestimmte Speicherstellen, um Variablen aufzubewahren oder Platz für einen späteren Zugriff zu reservieren. Der MASM 4.0 verfügt über eine Reihe von Pseudobefehlen, die einer Variablen einen bestimmten Wert oder Speicherbereich zuordnen. Ein Pseudobefehl für die Datendefinition muß nach folgendem Schema aufgebaut sein:

```
[Variable] Definitionsbefehl Operand
[,Operand [,Operand [,...]]]
```

Die Angaben in den eckigen Klammern sind optional. Für den oder die Operanden sind verschiedene Formate (Bitfeld, dezimal, hexadezimal etc.) zulässig.
Es ist vollkommen unterschiedlich, mit welchem Wert jeder Operand höchstens belegt werden darf, denn der Höchstwert richtet sich nach dem verwendeten Pseudobefehl. Der Pseudobefehl »db« erlaubt es Ihnen sogar, neben Zahlen auch Zeichenketten direkt in den Speicher zu schreiben. Die folgenden Beispiele zeigen Ihnen, wie man mit Hilfe der Pseudobefehle Konstanten definiert:

```
plz dw 8013
name db 'DOS International'
prozessor dd 80386
```

Neben der Daten- und Konstantendefinition sind Sie durch die Definitions-Pseudobefehle auch in der Lage, Tabellen aufzubauen, indem Sie zwei oder mehrere Operanden durch Kommata trennen. Das folgende Beispiel demonstriert solch einen Tabellenaufbau:

```
adresse   db 'Firma $'
          db 'IBM Deutschland $'
          db 'Postfach 800880 $'
          db 'Pascalstr. 100$'
          db '7000 Stuttgart 80 $'
tabelle   db 80, 83, 47, 50
          db 45, 77,111,100
          db101,108,108,101
```

Zum Reservieren von Speicherbereichen verwenden MASM 4.0 und SAM III unterschiedliche Befehle

Wichtig ist auch die Funktion der Definitions-Pseudobefehle zum Reservieren von Speicherbereichen. Hier unterscheiden sich der MASM 4.0 und SAM III. Um beispielsweise 255 Byte für einen Puffer zu reservieren, dessen Inhalt zu Beginn nicht feststeht, müssen Sie beim MASM 4.0 schreiben:

```
puffer db 255 dup (?)
```

Bei SAM III vereinfacht sich diese Befehlszeile zu

```
puffer ds 255
```

Der MASM 4.0 stellt für die Programmierer von größeren Software-Projekten, die sich beispielsweise für unterschiedliche Hardware-Konfigurationen assemblieren lassen müssen, eine Gruppe von bedingten Pseudobefehlen zur Verfügung. Bedingte Pseudobefehle veranlassen den Assembler, einen bestimmten Teil eines Quellprogramms zu assemblieren oder ihn zu ignorieren; je nachdem, ob eine Bedingung während des Übersetzungsvorgangs logisch wahr oder falsch ist.

Durch dieses wahlweise Assemblieren bestimmter Quelltextteile ist es leicht möglich, ein Programm an verschiedene Computersysteme oder Peripherie-Einheiten anzupassen. Besonders sinnvoll ist diese Einrichtung, wenn ein Assemblerprogramm Systemvariablen und Systemadressen benutzt, die vom Computermodell abhängig sind.
Falls Sie einen Teil Ihres Programms für die bedingte Assemblierung vorgesehen haben, so müssen Sie die gewünschten Befehlszeilen mit einem der in der Tabelle aufgeführten If-Befehle einleiten und mit dem Befehl »endif« abschließen. Erkennt der MASM 4.0 einen dieser Pseudobefehle, dann assembliert er die eingeschlossenen Anweisungen, falls der Ausdruck logisch wahr ist. Andernfalls ignoriert der Assembler die in dieser Weise gekennzeichneten Befehlszeilen und unterbricht den Übersetzungsvorgang, bis der Programmteil auftritt, der mit »else« beginnt. Ist kein Else-Befehl aufgeführt, dann fährt der Assembler bei der Anweisung fort, die auf das »endif« folgt. Das folgende Beispiel demonstriert den Aufbau einer If-Konstruktion:

```
if angeschlossen
drucken 13h,10h ;Makro-Aufruf Drucker
else
ausgabe,13h,10h
;Makro-Aufruf Bildschirm
endif
```

Bei jedem Übersetzungsvorgang, zu dem die vorher definierte Variable »angeschlossen« einen Wert ungleich Null enthält, wird das Makro »drucken« in den Quelltext eingefügt. Ist hingegen »angeschlossen« gleich Null, so wird das Makro »ausgabe« übersetzt.
Die Makropseudobefehle gehören zu den sinnvollsten Erweiterungen eines Assemblers. Die Hauptbefehle »macro« und »endm«, die den Anfang und das Ende eines Makros definieren, erlauben dem Assemblerprogrammierer den Aufbau eigener Makrobibliotheken, die das ständige Neuprogrammieren von Standardroutinen ersparen. So ist es zum Beispiel möglich, immer wiederkehrende Routinen mit unterschiedlichen Parametern als Makro zu definieren. Dieses Verfahren macht Assemblerprogramme übersichtlich und besser lesbar. Allerdings unterscheiden sich die Art der Definition und die Form des Aufrufs eines Makros bei MASM 4.0 und SAM III erheblich. Beim MASM 4.0 wird ein Makro mit der Übergabe von zwei Parametern zum Beispiel wie folgt eingegeben:

```
funktion  macro variable,nummer
          mov dx,variable
          mov ah,nummer
          int 21h
endm
```

Der Aufruf erfolgt durch die bloße Angabe der Makrobezeichnung:

```
funktion text,09h
```

Der MASM 4.0 ersetzt beim Assemblieren den Makroaufruf durch das Makro selbst, wobei er »variable« gegen »text« und »nummer« gegen den Wert 09hex austauscht.
SAM III behandelt Makrodefinition und Makroaufruf etwas anders. Jeder Parameter wird bis zu einer Höchstzahl von acht Angaben durch eine Kombination aus dem Prozentzeichen und einer Ziffer repräsentiert. Das oben beschriebene Beispiel für MASM 4.0 lautet bei SAM III folgendermaßen:

```
funktion  macro %1,%2
          mov dx,%1
          mov ah,%2
          int 21h
endm
```

Der Makroaufruf erfolgt bei SAM III ebenfalls etwas anders, als es bei MASM 4.0 der Fall ist:

```
do funktion,text,09h
```

MASM 4.0 besitzt darüber hinaus eine Reihe weiterer, leistungsfähiger Makrobefehle, die in vielen Situationen sehr hilfreich sind. Beispielsweise kann man mit »exitm« ein Makro vorzeitig verlassen, wenn der Befehl im Zusammenhang mit der bedingten Assemblierung benutzt wird. Falls ein Makro Sprungmarken enthält, ist es wünschenswert, daß der Assembler diese Marken von denen des Hauptprogramms unterscheidet. Andernfalls träte in jedem Fall der Fehler »Symbol mehrfach definiert« auf. Um diese Fehlermeldung zu vermeiden, kann der Programmierer dem MASM mit dem Pseudobefehl »local« mitteilen, welche Sprungmarken ausschließlich lokal zu verwenden sind. Dabei ist es wichtig, daß »local« am Anfang eines Makros steht. Die Befehlsfolge

```
befehl    macro eins,zwei
          local schleife
schleife: mov ax,eins
          inc zwei
          jpe schleife
          endm
          ;
schleife: befehl drucken,ax
```

ist damit durchaus legitim.

Der MASM verarbeitet zusätzlich die drei Makropseudobefehle »rept«, »irp« und »irpc«, die den Assembler veranlassen, eine Folge von Anweisungen mehrmals zu wiederholen. Bei jedem dieser Befehle wird die Anzahl der Wiederholungen im Operandenfeld angegeben. Der Operand des Pseudobefehls »rept« muß dabei ein mathematischer Ausdruck oder ein Symbol sein, aus dem sich die Anzahl der Wiederholungen ermitteln läßt. Beim Pseudobefehl »irp« können Sie zusätzlich zu einem anzugebenden Hilfsparameter eine Liste bestimmen. Die Anweisung wird dann für jedes Element wiederholt. Das folgende Beispiel erläutert die Funktion dieser Pseudobefehle anhand des Befehls »rept«. Das kurze Listing gibt alle ASCII-Zeichen aus:

```
ascii macro
code= 0
rept256
callausgabe
code= code + 1
endm
```

Der letzte Makrobefehl, der bei MASM 4.0 Verwendung findet, heißt »purge«. Dieser Pseudobefehl löscht Makros aus den internen Tabellen des Assemblers, so daß der von ihnen belegte Arbeitsspeicher wieder frei wird. Der Pseudobefehl löscht aber keineswegs die Makroquelle oder das Objektprogramm, das sich auf der Diskette oder Festplatte befindet. SAM III umgeht dieses Problem, indem er alle Makros auf Diskette oder Festplatte auslagert und nur bei Bedarf wieder lädt.
Wenn Ihnen eine Zusammenstellung mehrerer Makros (beispielsweise eine Makrobibliothek) als eigenständige Datei vorliegt, die Sie in einem anderen Programm verwenden möchten, müssen Sie natürlich dafür sorgen, daß der jeweilige Assembler beim Übersetzen Ihres Programms diese externen Dateien auch in den Assemblerquelltext einbindet. Das geschieht mit dem Pseudobefehl »include«, der sowohl den MASM als auch SAM III veranlaßt, die benötigten Dateien einzulesen und zu verwerten. Dazu müssen Sie hinter dem Include-Befehl nur den Namen der Datei aufführen, die die gewünschten Makros enthält. Sollen mehrere Makrodateien verwendet werden, so benutzen Sie den Befehl entsprechend mehrmals im Programm. Der Pseudobefehl »include« dient aber ebenso dazu, schon vorhandene und auf Lauffähigkeit getestete Quelltexte in das Hauptprogramm einzubinden. Die Syntax bleibt in diesem Fall gleich.

Der Include-Befehl erlaubt das Einbinden bereits bestehender Quelltexte in ein neues Programm

Möchten Sie beispielsweise eine Ausgaberoutine namens »drucker.asm«, die sich auf der Diskette im Laufwerk A befindet, in Ihr Hauptprogramm »ausgabe.asm« eingliedern, so setzen Sie die Befehlszeile

```
include a:drucker.asm
```

zu Beginn von »ausgabe.asm« ein.

Wenn der Assembler dann das Hauptprogramm übersetzt, liest er alle Befehlszeilen aus der Datei »drucker.asm« ein und behandelt sie, als würden sie direkt an der Position des zugehörigen Include-Befehls im Quelltext stehen.

Der Pseudobefehl »include« gehört wie die MASM-Befehle »extrn« und »public« zu der Gruppe der Pseudobefehle für externe Verweise. Mit diesen Befehlen können Sie auf Informationen verweisen, die extern vorhanden sind. Mit »public« werden die angegebenen Symbole auch für andere Assemblermodule zugänglich, die eventuell zum aktuellen Modul hinzugeladen werden. Der Pseudobefehl »extrn« indiziert hingegen die Symbole, die in anderen Modulen definiert und als »public« deklariert wurden. Wollen Sie beispielsweise die Variable »zaehler« in zwei verschiedenen Modulen verwenden, so muß der Teil, in dem »zaehler« definiert wird, folgende Struktur aufweisen:

```
          public zaehler
zaehler   db 0 ; Zähler auf Null setzen
```

Um »zaehler« in dem anderen Modul zu verwenden, geben Sie folgende Anweisung ein, die auch den Variablentyp (»byte«, »word«, »dword«, »far« oder »near«) bestimmt:

```
extrn zaehler:byte
```

Anstelle eines Makros läßt sich in einem Assemblerprogramm auch eine Prozedur einsetzen, die beim MASM 4.0 und bei SAM III durch den Pseudobefehl »proc« eingeleitet und mit »endp« beendet wird. Bei jedem Aufruf einer solchen Prozedur führt der Prozessor die Befehle aus, die durch »proc« und »endp« eingegrenzt sind, und kehrt darauf zu dem Befehl zurück, der dem Prozeduraufruf folgt. Eine typische Prozedur ist etwa:

```
ausgabe proc
:
ret
ausgabe endp
```

Im Gegensatz zu einem Makro muß eine Prozedur jedesmal über den Befehl »call« aufgerufen und mit »ret« beendet werden. Das kostet Zeit, und die ist – vor allem für den Assemblerprogrammierer – äußerst kostbar.

Nach dem Assemblieren erscheint der Maschinencode einer Prozedur nur ein einziges Mal im Programm, während der Maschinencode eines Makros überall dort vorhanden ist, wo das Makro benötigt wurde. Aus diesem Grund begnügt sich eine Prozedur mit erheblich weniger Speicherplatz als ein Makro; besonders dann, wenn das Makro sehr lang ist und oft benutzt wird. Kurz gesagt, eine Prozedur spart Speicherplatz und ein Makro Zeit.

Doch das Makro hat noch einen wesentlichen Vorteil, und der liegt in der Dynamik. Ein Makro kann nämlich bei jedem Aufruf leicht verändert werden, indem die übergebenen Parameter verändert werden. Prozeduren übernehmen Parameter dagegen nur in Registern und Speicherzellen, wodurch sie ziemlich unflexibel sind.

Ein weitere Gruppe von Pseudobefehlen ist die der Ausgabesteuerung. Weil die in dieser Gruppe enthaltenen Befehle jedoch sehr zahlreich sind, werden im folgenden nur die drei wichtigsten erläutert.

Der Pseudobefehl »page« hat den Aufbau

```
page [zeilen] [,spalten]
```

und legt das Druckformat bei der Ausgabe des übersetzten Quelltextes auf einen Drucker fest. Die Parameter »zeilen« und »spalten« definieren die Länge und Breite der einzelnen Seiten. Der Pseudobefehl »title« versieht jede zweite Zeile einer ausgegebenen Seite mit einer linksbündigen Überschrift. In den meisten Fällen beinhaltet diese Überschrift den Dateinamen, Sie können sie aber frei wählen. Mit »subttl« wird schließlich in der dritten Zeile jeder Seite ein linksbündiger Untertitel ausgegeben. Auch er ist frei wählbar und beschreibt im allgemeinen den Inhalt der entsprechenden Seite. Die folgenden Pseudobefehle definieren eine typische Kopfzeile:

```
page 66,80
title framework.asm
subttl Die Textverarbeitung
```

Die letzte Gruppe der Pseudobefehle steuert einen Assembler direkt an. Der wichtigste Befehl, der garantiert in jedem Assemblerprogramm auftritt, ist »end«. Dieser Pseudobefehl markiert das Ende eines Quelltextes und teilt dem Assembler mit, an welcher Stelle er die Assemblierung beenden kann. Der End-Befehl muß aus diesem Grund in jedem Assemblerprogramm vorhanden sein. MASM 4.0 verarbeitet darüber hinaus die Befehlsform

```
end marke
```

die mit der Sprungmarke »marke« die Startadresse für den assemblierten Maschinencode angibt. SAM III nimmt hingegen an, daß der Anfang des Codesegments der Startadresse des Programms entspricht.

Der Pseudobefehl »org« setzt bei einem Assembler den Adreßzähler neu, der bestimmt, unter welchen Adressen die Befehle und Daten im Arbeitsspeicher abgelegt werden. Beim PC übernimmt zwar normalerweise das Betriebssystem MS-DOS diese Entscheidung, aber es ist in gewissen Fällen notwendig, diese zu beeinflussen.

Diese Notwendigkeit tritt dann ein, wenn ein Maschinensprachepro-gramm als Datei mit der Endung »com« gespeichert werden soll. In diesem Fall muß vor dem ersten Befehl der Adreßzähler mit

```
org 100h
```

auf den Wert 100hex gesetzt werden, weil MS-DOS in den ersten 100hex Byte wichtige Systeminformationen ablegt. Der Assemblercode läßt sich auch relativ zum aktuellen Adreßzähler übersetzen, weil der aktuelle Zustand des Adreßzählers durch das Dollarzeichen repräsentiert wird. Das bedeutet, daß beispielsweise die Anweisung

```
org $ + 1024
```

den Adreßzähler um den Wert 1024 erhöht.

Unmittelbar im Zusammhang mit »org« steht der Pseudobefehl »even«. Er erzwingt, daß der Adreßzähler auf eine gerade Adresse gesetzt wird. Weist der Adreßzähler bereits einen geraden Wert auf, so passiert gar nichts. Ist die Adresse dagegen ungerade, so fügt der Assembler beim Übersetzen den Dummy-Assemblerbefehl »nop« ein, der dafür sorgt, daß der Adreßzähler wieder auf eine gerade Adresse zeigt, aber selbst keine Aktion bewirkt. Der Einsatz des Pseudobefehls »even« ist vor allem dann sinnvoll, wenn Programme speziell für den 8086-Prozessor entwickelt werden. Da dieser auch nur 16 Datenbits gleichzeitig übertragen kann, dauert dieser Vorgang länger, wenn die übertragenen Daten auf einer ungeraden Adresse liegen. So profitieren vor allem zeitkritische Programme von dem Einsatz des Even-Befehls.

Zu guter Letzt sei der Pseudobefehl ».radix« erwähnt. Bekanntlich lassen sich Daten binär, dezimal und hexadezimal eingeben. Um die einzelnen Zahlensysteme voneinander zu unterscheiden, gibt ein Identifikationsbuchstabe die jeweilige Zahlenbasis an, wenn sie von der dezimalen Standardbasis 10 abweicht.

Tritt jedoch der Fall ein, daß Sie mit vielen nichtdezimalen Zahlen arbeiten müssen, empfiehlt es sich, die Standardbasis Ihren Erfordernissen entsprechend anzupassen. Und genau dazu dient der Pseudobefehl ».radix«; Sie können damit bei MASM 4.0 die Standardbasis von 10 auf jede andere Basis von 2 bis 16 umstellen. Die Befehlszeile

```
.radix 16
```

legt beispielsweise die Zahlenbasis neu auf 16 fest, so daß Sie fortan mit Hexadezimalzahlen arbeiten können, ohne an jede Zahl den Buchstaben »h« anhängen zu müssen.

(Markus Zietlow/al)

MASM	SAM III	Erklärung
		Ausgabesteuerung:
PAGE	—	Stellt die Länge und Breite einer Ausdruckseite ein
SUBTTL	SUBTTL	Definiert den Untertitel eines Quelltextausschnittes (wird von SAM III ignoriert)
TITLE	TITEL	Definiert den Titel eines Listings (wird von SAM III ignoriert)
.CREF	—	Gegenstück zu ».XCREF«
.LALL	—	Bewirkt bei der Ausgabe des Assemblerlistings, daß alle Makrotexte des Haupttextes und jeder Erweiterung ausgegeben werden
.LFCOND	—	Listet alle bedingten Blöcke auf, die als logisch falsch ausgewertet wurden
.LIST	—	Gegenstück zu ».XLIST«
.OUT	—	Veranlaßt die Ausgabe von Systemmeldungen auf dem Bildschirm während des Assemblierens
.SALL	—	Unterdrückt die Ausgabe der Makros
.SFCOND	—	Listet als falsch ausgewertete bedingte Blöcke nicht auf
.TFCOND	—	Die Wirkung der letzten Anweisung .LFCOND bzw. .SFCOND wird umgekehrt
.XALL	—	Gibt in bezug auf Makros ausschließlich die Zeilen aus, die Objektcode erzeugen
.XCREF	—	Unterdrückt die Ausgabe der Querverweisliste («Cross Reference«) von der aktuellen Position des Pseudobefehls bis zum nächsten Befehl ».CREF«
.XLIST	—	Unterdrückt die Ausgabe des Assemblerlistings von der aktuellen Position des Pseudobefehls bis zum nächsten ».LIST«
		Bedingte Assemblierung:
ELSE	—	Gibt an, daß der folgende Befehlsblock ausgeführt werden soll, wenn der vorangegangene Vergleich logisch falsch war
ENDIF	—	Angabe des Endes einer bedingten Assemblierung
IF	—	Ergibt den logischen Wert »wahr«, wenn der angegebene Ausdruck ungleich Null ist
IFB	—	Ergibt den logischen Wert »wahr«, wenn das Argument leer ist
IFDEF	—	Ergibt den logischen Wert »wahr«, wenn das spezifizierte Symbol in- oder extern definiert wurde
IFE	—	Ergibt den logischen Wert »wahr«, wenn der Ausdruck gleich Null ist
IFIDN	—	Ergibt den logischen Wert »wahr«, wenn die beiden angegebenen Zeichenketten identisch sind
IFIDF	—	Ergibt den logischen Wert »wahr«, wenn die beiden angegebenen Zeichenketten sich unterscheiden
IFNB	—	Ergibt den logischen Wert »wahr«, wenn das Argument nicht leer ist
IFNDEF	—	Ergibt den logischen Wert »wahr«, wenn das spezifizierte Symbol in- oder extern nicht definiert wurde
IF1	—	Ergibt den logischen Wert »wahr«, wenn der Assembler den ersten Durchlauf durchführt
IF2	—	Ergibt den logischen Wert »wahr«, wenn sich der Assembler im zweiten Durchgang befindet
		Datendefinition:
DB	DB	Definiert Byte-Speicherbereich für Daten (8 Bit); es lassen sich mehrere Werte zuweisen
DD	—	Definiert Doppelwort-Speicherbereich für Daten (32 Bit); es lassen sich mehrere Werte zuweisen
DQ	—	Definiert Vierfachwort-Speicherbereich für Daten (64 Bit); es lassen sich mehrere Werte zuweisen
—	DS	Reserviert die angegebene Anzahl von Bytes für die spätere Verwendung (entspricht dem MASM-Pseudobefehl »DB anzahl DUP (?)«)
DT	—	Weist zehn Bytes zu (wird hauptsächlich in der Fließkomma-Arithmetik benutzt)
DW	—	Definiert Wort-Speicherbereich für Daten (16 Bit); es lassen sich mehrere Werte zuweisen
		Externe Informationen:
EXTRN	—	Gibt Symbole an, die in anderen Modulen definiert wurden; so kann auf bereits überprüfte Assemblerquelltexte mit Querverweisen zugegriffen werden
INCLUDE	INCLUDE	Fügt den Quelltext mit dem angegebenen Dateinamen in das Listing ein; auf diese Weise können Sie Bibliotheken bequem einbinden
PUBLIC	—	Kennzeichnet die Symbole, die andere Assemblerprogramme verwenden können, nachdem Sie verknüpft worden sind
		Makrodefinition:
—	DO	Aufruf eines zuvor definierten Makros. Der folgende Parameter gibt den Makronamen an, die weiteren Parameter werden in das Makro übernommen
ENDM	ENDM	Schließt eine Makrodefinition ab und kennzeichnet das Makroende
EXITM	—	Beendet die Ausführung eines Makros aufgrund eines vorangegangenen Vergleichs
IRP	—	Wiederholt die Anweisungen zwischen »IRP« und »ENDM« für jedes Argument in der angegebenen Argumentliste mit den einzelnen Elementen
IRPC	—	Enspricht grundsätzlich »IRP«; der Unterschied besteht darin, daß statt der Argumentliste eine Zeichenkette übergeben wird
LOCAL	—	Bewirkt, daß der Assembler jede mit diesem Befehl definierte Marke als lokal auffaßt, was bedeutet, daß in diesem Fall das Hauptprogramm die gleichen Marken enthalten kann
MACRO	MACRO	Definiert die vorangehende Marke als Namen eines Makros, das wiederum durch die nachstehenden Parameter definiert wird
PURGE	—	Löscht das angegebene Makro aus dem Arbeitsspeicher des Computers, so daß der benutzte Speicher wieder verwendet werden kann
REPT	—	Wiederholt alle Anweisungen, die sich zwischen »REPT« und »ENDM« befinden, so lange, bis die angegebene Anzahl erreicht ist
		Programmblockangabe:
GROUP	—	Faßt mehrere Segmente unter einem Namen zusammen, so daß sie alle in einem 64-KByte-Bereich innerhalb des Arbeitsspeichers liegen
NAME	—	Weist einem Modul den vorangehenden Namen zu
		Segmentangaben:
ASSUME	—	Teilt dem Assembler mit, welches Segment den angegebenen Segmentregistern zugeordnet ist
ENDS	ENDS	Markiert das Ende eines Segments
SEGMENT	SEGMENT	Kennzeichnet den Beginn eines Speichersegments
		Symboldefinition:
EQU	EQU	Weist einem symbolischen Namen einen konstanten Wert zu
=	=	Entspricht dem Pseudobefehl »EQU«
		Unterprogrammangabe:
ENDP	ENDP	Markiert das Ende eines Unterprogramms
PROC	PROC	Kennzeichnet den Beginn eines Unterprogramms
		Weitere Befehle:
END	END	Gibt das Ende des Assemblerprogramms an
EVEN	EVEN	Setzt den Befehlszähler auf eine gerade Adresse
ORG	ORG	Setzt den Befehlszähler auf die angegebene Adresse (beispielsweise für COM-Dateien notwendig)
.RADIX	–	Ändert die Standardbasis bei der Angabe von Zahlen auf jede beliebige Basis von 2 bis 16

Tabelle. Die Pseudobefehle der Makroassembler MASM 4.0 und SAM III thematisch geordnet

Das Einmaleins des Prozessors

Die Arithmetik auf Maschinensprachebene ist ein Thema, das selbst fortgeschrittenen Assemblerprogrammierern gelegentlich Schwierigkeiten bereitet. Die Zahlen müssen in Registern »gehortet« werden, das Carry-Flag ist zu beachten, daneben muß man die umgekehrte Notation verstehen, binär addieren und vieles mehr. Damit Sie nicht länger alles mit den Fingern abzählen müssen, lernen Sie im folgenden die Grundlagen der Prozessor-Arithmetik kennen.

Dieser Beitrag stellt Ihnen die Techniken der Assemblerprogrammierung für Arithmetikroutinen mit dem Schwerpunkt Integer-Arithmetik vor. Aufbau und Funktion der Arithmetikbefehle des 8086/8088-Prozessors werden erläutert und die zugrundeliegenden mathematischen Operationen für ganze Zahlen, die den Umfang der Arithmetikbefehle des Prozessors übersteigen, werden vorgestellt. Arithmetikroutinen bestehen ausschließlich aus Befehlen für die Grundrechenarten Addition, Subtraktion, Multiplikation und Division, weil der 8086/8088-Prozessor nur Befehle für diese mathematischen Operationen beherrscht. Alle anderen arithmetischen und auch trigonometrischen Funktionen werden von diesen Befehlen abgeleitet.

Das Carry-Statusbit ist bei der Addition von größeren Zahlen besonders wichtig

Die Routinen, die im folgenden vorgestellt werden, arbeiten mit ganzzahligen Werten. Der 8086/8088-Prozessor legt diese Werte in umgekehrter Reihenfolge im Arbeitsspeicher ab. Wenn der Prozessor beispielsweise die hexadezimale Zahl 4782hex in der Speicherzelle »data« speichert, steht der Wert 82hex in »data« und der Wert 47hex in »data + 1«.
Dieses Verfahren der umgekehrten Speicherung, auch als Intel-Format bekannt, müssen Sie bei allen Operationen berücksichtigen, die mit Werten arbeiten, die aus mehreren Bytes oder Worten bestehen.
Der 8086/8088 führt arithmetische Operationen mit binären und dezimalen Werten ohne Vorzeichen durch. Zu den wichtigsten Befehlen gehören die Additionsbefehle, die 8- und 16-Bit-Operanden zusammenzählen. Der Befehl

```
add AX,BX
```

addiert beispielsweise die Inhalte der beiden 16-Bit-Register AX und BX und legt das Ergebnis in AX ab. Allerdings gibt es dann ein (scheinbares) Problem, wenn der Zieloperand, in diesem Fall das AX-Register, die Summe der beiden Werte nicht aufnehmen kann. Weil das Problem des »Fassungsvermögens« natürlich auch bei 8-Bit-Registern zutage treten kann, sei es an einem Beispiel demonstriert: Sie zählen die beiden folgenden 8-Bit-Register zusammen:

```
  AL: 10101010
+ BL: 11010101
=     101111111
```

Weil für die Darstellung dieses Ergebnisses ein 8-Bit-Register nicht ausreicht, schreibt der Prozessor die niederwertigen acht Bit in das Zielregister und legt das neunte Bit als Übertrag im Carry-Statusbit ab.
Befindet sich nun in den Registern AL und BL jeweils nur die untere Hälfte der beiden zu addierenden Werte und in den Registern AH und BH die obere Hälfte, so können Sie diese mit dem Befehl »adc« zusammenzählen, der den Übertrag im Carry-Statusbit berücksichtigt.
Viel einfacher wird es natürlich, wenn Sie statt dessen gleich den 16-Bit-Befehl

```
add AX,BX
```

benutzen. Doch wenn Sie Werte addieren möchten, die größer als zwei Byte sind, hat das Verfahren mit Übertrag und dem Additionsbefehl »adc« für alle höherwertigen Bytes wieder Sinn.
Analog zu den Additionsbefehlen gibt es auch bei den Subtraktionskommandos zwei Varianten. Der Befehl »sub« zieht einen Subtrahenden von einem Minuenden ab und legt das Ergebnis in dem Register ab, in dem sich der Minuend befindet. So vermindert beispielsweise der Befehl

```
sub AX,BX
```

den Inhalt von AX um den Inhalt von BX und legt das Ergebnis im AX-Register ab.
Das Kommando »sbb« berücksichtigt bei der Subtraktion zusätzlich einen eventuell vorhandenen Übertrag im Carry-Statusbit und bezieht diesen in die Berechnung mit ein.
Auch für die Multiplikation besitzt der 8086/8088-Prozessor zwei verschiedene Befehle. Sie lauten:

```
mul Operand
```

und

```
imul Operand
```

Beide Befehle multiplizieren den angegebenen Operanden bei Byte-Operationen (8 Bit) mit dem AL- und bei Wort-Operationen (16 Bit) mit dem AX-Register. Im ersten Fall wird das Ergebnis in AX und im zweiten in DX und AX zurückgegeben, wobei das DX-Register das höherwertige Wort enthält.

Einer der Multiplikationsbefehle berücksichtigt das Vorzeichen

Die beiden Multiplikationsbefehle unterscheiden sich darin, daß der Befehl »imul« im Gegensatz zu »mul« bei der Ausführung ein eventuell vorhandenes Vorzeichen berücksichtigt. Entsprechend zu den beiden Multiplikationsbefehlen verarbeitet der 8086/8088 auch zwei verschiedene Divisionsbefehle. Die beiden Befehle lauten

```
div Divisor
```

und

```
idiv Divisor
```

Der Befehl »div« teilt zwei Zahlen ohne Berücksichtigung des Vorzei-

chens, und »idiv« dividiert entsprechend zwei Werte unter Berücksichtigung des Vorzeichens. Sie können allerdings jeweils nur den Divisor (die Zahl, durch die die andere geteilt wird) spezifizieren. Als Dividenden (die Zahl, die durch den Divisor geteilt werden soll) verwendet der 8086/8088 automatisch das Register AX bei Byte-, sowie DX und AX bei Wort-Operationen.

Liegt der Divisor im Byteformat vor, so wird das Ergebnis der Division im AL- und der Rest im AH-Register zurückgegeben. Ist der Divisor dagegen 16 Bit groß, so speichert der Prozessor das Ergebnis in AX und den Rest in DX ab.

Der 8086/8088-Prozessor ist in der Lage, mit einem einzigen Befehl eine Division durchzuführen

Beachten Sie bei allen Additions-, Subtraktions-, Multiplikations- und Divisionsbefehlen, daß sich weder zwei Zahlen direkt noch zwei Werte im Speicher unmittelbar miteinander verknüpfen lassen. Vielmehr muß sich zumindest einer der beiden Werte in einem Register befinden, in das er über einen einfachen Mov-Befehl geladen wird. Bei den Multiplikations- und Divisionsbefehlen sind sogar ganz bestimmte Register vorgeschrieben.

Bei den Divisionsbefehlen tritt eine weitere Besonderheit auf. Falls nämlich das Ergebnis einer Division die Kapazität des Zielregisters überschreitet oder der Divisor den Wert Null enthält, ruft der Prozessor automatisch den Interrupt 0 (»Division durch 0«) auf. Wenn also ab der Speicheradresse 0 ein Interruptvektor, bestehend aus einer 16-Bit-Offset- und Segmentadresse abgelegt ist, der auf eine Fehlerbehandlungsroutine zeigt, wird diese Routine automatisch aufgerufen, sobald bei der Division einer der beiden erwähnten Fehler auftritt.

Neben den vorgestellten grundlegenden Arithmetikbefehlen gibt es einige weitere, die nur in speziellen Fällen Anwendung finden. Die Funktion dieser und der bisher behandelten Arithmetikbefehle entnehmen Sie der Tabelle 1.

Die Arithmetikbefehle des 8086/8088-Prozessors reichen für viele Anwendungen aus. Wenn Sie jedoch mit größeren Zahlen oder Fließkommaroutinen arbeiten wollen, müssen Sie sich mit selbstgeschriebenen Programmen behelfen. Wie solche Arithmetikroutinen aufgebaut sind, und wie Sie die Werte verarbeiten, die die Größe eines Registers übertreffen, erfahren Sie im folgenden.

Befehl	Funktion
AAA	Passe Ergebnis nach einer ASCII-Addition an
AAD	Passe Ergebnis nach einer ASCII-Division an
AAM	Passe Ergebnis nach einer ASCII-Multiplikation an
AAS	Passe Ergebnis nach einer ASCII-Subtraktion an
ADC	Addiere unter Berücksichtigung des Übertrags
ADD	Addiere ohne Übertrag
CBW	Umwandlung eines Bytes in ein Wort
CMP	Führe einen Vergleich der angegebenen Operanden aus
CWD	Umwandlung eines Worts in ein Doppelwort
DAA	Passe Ergebnis nach einer dezimalen Addition an
DAS	Passe Ergebnis nach einer dezimalen Subtraktion an
DEC	Vermindere angegebenen Operanden um den Wert 1
DIV	Führe vorzeichenlose Division aus
IDIV	Dividiere Operanden unter Berücksichtigung des Vorzeichens
IMUL	Multipliziere Operanden mit Vorzeichen
INC	Erhöhe angegebenen Operanden um den Wert 1
MUL	Führe vorzeichenlose Multiplikation aus
NEG	Negiere Operanden beziehungsweise bilde sein Zweierkomplement
SBB	Subtrahiere unter Berücksichtigung des Übertrags
SUB	Subtrahiere ohne Übertrag

Tabelle 1. Die Arithmetikbefehle des 8086/8088-Prozessors im Überblick

Die Addition, als einfachste Grundrechenart, erleichtert den Anfang. Wie bereits ausgeführt, muß hierbei vor allem berücksichtigt werden, ob bei der Addition zweier Zahlen ein Übertrag auftreten kann. Falls ein Übertrag entsteht, so wird dieser durch das gesetzte Carry-Statusbit dokumentiert und muß entsprechend zu Rate gezogen werden. Wenn Sie Zahlen addieren, die aus mehreren Bytes bestehen, werden zuerst die beiden niederwertigsten Bytes mit »add« und alle weiteren Bytes mit »adc« addiert, damit bei den höherwertigen Bytes jeweils der Übertrag berücksichtigt wird.

Probleme beim Multiplizieren gibt es erst bei Zahlen, die größer als 16 Bit sind

Die Subtraktion von Zahlen, die aus mehreren Bytes bestehen, entspricht dem Vorgang bei der Addition. Hier wird beim Subtrahieren der beiden niederwertigsten Bytes der Befehl »sub« und bei den höherwertigen Bytes jeweils der Befehl »sbb« benutzt, um einen eventuell vorhandenen Übertrag im Carry-Statusbit einzubeziehen.

Bei der Multiplikation von Werten größer als 16 Bit wird die Berechnung schon komplizierter. Wie Sie bereits wissen, können Sie mit Hilfe des Befehls »mul« nur Werte bis zu einer maximalen Länge von 16 Bit miteinander multiplizieren. Bei den meisten praktischen Anwendungen muß man aber Zahlen multiplizieren, die 32 Bit oder länger sind; das bedeutet, daß die vorhandene Genauigkeit nicht ausreicht.

Listing 1 zeigt deshalb eine Assemblerroutine, die zwei 32-Bit-Werte miteinander multipliziert, ohne dabei die Vorzeichen zu beachten. Wenn Sie das grundlegende Prinzip dieser Routine erkannt haben, steht einer Erweiterung auf 48, 64 oder gar 80 Bit nichts mehr im Wege:

Die beiden Operanden für die Multiplikationsroutine sind jeweils in den Registerpaaren CX:BX und DX:AX abgelegt. (In dieser Notation kennzeichnet das Register vor dem Doppelpunkt das höherwertige und das Register nach dem Doppelpunkt das niederwertige Wort).

Das Ergebnis der Multiplikation wird nach Beendigung der Berechnung in die gleichen Register geschrieben, und zwar so, daß das DX-Register das höchstwertige und das AX-Register das niederwertigste Wort enthält. Das Bild zeigt, nach welchem Verfahren die Multiplikationsroutine die Inhalte der vier Register zu einem 64-Bit-Wert multipliziert.

```
(DX:AX) * (CX:BX)  =
-------------------------------------------------
AX * BX =                        speicher1:speicher2
DX * BX =              speicher3:speicher4
AX * CX =              speicher5:speicher6
DX * CX = speicher7:speicher8
-------------------------------------------------
        =    DX    :   CX    :   BX    :   AX
```

Nach diesem Schema arbeitet die Multiplikationsroutine in Listing 1

dezimal	binäre Mantisse	Exponent
24	00001100	1
24	00011000	0
24	00110000	-1
24	01100000	-2
24	11000000 (linksbündig)	-3

Tabelle 2. Die linksbündige Exponentialdarstellung ist der Standard

Beachten Sie dabei, daß die Routine keine Datenregister sichert. Das Sichern dieser Register muß deshalb vom aufrufenden Hauptprogramm erfolgen oder in die Multiplikationsroutine eingefügt werden. Mit diesen Ergänzungen ist die Routine ein wichtiges Unterprogramm, das die Grundlage für weitere Multiplikations- und Potenzfunktionen bildet.

Die Divisionsroutine in Listing 2 demonstriert, wie ein Überlauf abgefangen wird, der bei Divisionen auftritt, wenn der Dividend mindestens 65536mal so groß wie der Divisor ist. Dazu wird eine neue Routine für den Interrupt 0 (»Division durch 0«) installiert und der Dividend mit einer einfachen, aber etwas zeitaufwendigeren Prozedur durch den Divisor geteilt.

Die Routine dividiert einen 32-Bit-Wert im Registerpaar DX:AX durch einen 16-Bit-Wert in BX. Am Ende der Routine enthält das Registerpaar BX:AX das Ergebnis und DX den Rest der Division. Beachten Sie bei der Divisionsroutine auch den Sonderfall, der dann eintritt, wenn der Divisor gleich Null ist. In diesem Fall wird nämlich die alte Routine des Interrupt 0 aufgerufen, weil die neue Routine zu diesem Zeitpunkt noch nicht installiert ist.

Die Anregungen zur Implementierung neuer Routinen für die vier Grundrechenarten zeigen, daß man mit wenig Aufwand zu Berechnungen mit erhöhter Leistung und Genauigkeit gelangt. Bei Additionen, Subtraktionen und Verschiebungen empfiehlt es sich zusätzlich, die einzelnen Statusbits etwas genauer zu betrachten.

Fließkommazahlen arbeiten mit einem Exponenten und einer Mantisse

Die Berechnung von ganzen Zahlen mit Hilfe der Maschinenspracharithmetik ist nicht besonders schwierig. Wie ist es aber, wenn man mit Fließkommazahlen arbeiten will? Dieser Teilbereich der Mathematik ist für Maschinenspracheprogrammierer der anspruchvollste.

Eine Fließkommazahl besteht grundsätzlich aus zwei Bestandteilen: der Mantisse und dem Exponenten. Die Mantisse ist eine vorzeichenlose Binärzahl, deren Größe die Genauigkeit einer Zahl bestimmt. Sie hat üblicherweise eine Länge von 24, 32, 48 oder 64 Bit. Der Exponent ist eine vorzeichenbehaftete Binärzahl mit ebenfalls vorgegebener Länge, deren Größe den Wertebereich der Zahl festlegt. Er ist normalerweise ein bis zwei Byte groß. Wenn das Vorzeichen einer Fließkommazahl gesondert abgelegt wird, läßt sie sich in der Form

$$\text{Mantisse} \times 2^{\text{Exponent}}$$

darstellen. In vielen Fällen gibt es jedoch verschiedene Variationen, um die gleiche Zahl im Fließkommaformat anzugeben. So können Sie beispielsweise die Zahl 24 in den folgenden Exponentialformaten notieren:

$$24 = 3 \times 2^3 = 6 \times 2^2 = 12 \times 2^1 = 24 \times 2^0$$

Das standardisierte Format verwendet die linksbündige Notation

Um ein standardisiertes Format zu erhalten, verwendet man die sogenannte »linksbündige Notation«, die den kleinsten Exponenten und die größtmöglichste Mantisse verkörpert. Ein mit der natürlichen Zahl 24 durchgeführtes Beispiel finden Sie in Tabelle 2. Der Wert der Zahl ändert sich – wie Sie sehen – bei der mathematischen Operation nicht.

Wenn Sie zwei Zahlen im Fließkommaformat addieren wollen, müssen Sie zuerst ihre Exponenten angleichen. Bei gleichen Exponenten vereinfacht sich die Addition nämlich wesentlich, und es gilt folgende Formel:

$$\text{Mantisse1} \times 2^{\text{Exponent}} + \text{Mantisse2} \times 2^{\text{Exponent}} = (\text{Mantisse1}+\text{Mantisse2}) \times 2^{\text{Exponent}}$$

Das heißt, nach der Exponentenangleichung können Sie die beiden Mantissen einfach addieren.

Der Unterschied der Subtraktion zur Addition besteht darin, daß beim Abziehen das Ergebnis zusätzlich noch negiert werden muß. Eine Zahl negieren Sie, indem Sie ihr Bitmuster invertieren und danach die Zahl Eins addieren. Eine Beispielrechnung finden Sie in Tabelle 3.

Diese Grundlagen stellen (beinahe) das ganze Geheimnis der Fließkommazahlen dar. Denn wenn beispielsweise eine Dezimalzahl mit Komma vorliegt, müssen Sie diese Zahl nur so lange mit zehn multiplizieren, bis sie kein Komma mehr enthält. Nach der mathematischen Operation dividieren Sie die errechnete Zahl entsprechend oft durch zehn, um das korrekte Ergebnis zu erhalten. Beachten Sie, daß bei dieser Art der Fließkommadarstellung bei sehr kleinen Zahlen Rundungsfehler auftreten, die in den meisten Fällen aber aufgrund Ihrer Geringfügigkeit nicht ins Gewicht fallen.

Mit den theoretischen Kenntnissen dieses Artikels bewaffnet, sind Sie nun gegen die ersten Angriffe von Zahlenkolonnen gefeit.

(Markus Zietlow/al)

Addition von Binärzahlen (12 + 39):

Operation	Mantisse 1	Exponent 1	Mantisse 2	Exponent 2
Ausgangszahlen	00001100	0	00100111	0
linksbündig anordnen	11000000	-4	10011100	-2
Exponenten angleichen	00110000	-2	10011100	-2
Mantissen addieren	11001100	-2		

Ergebnis: $204 \times 2^{-2} = 204 \times 1/4 = 51$

Subtraktion von Binärzahlen (12 - 39):

Operation	Mantisse 1	Exponent 1	Mantisse 2	Exponent 2
Ausgangszahlen	00001100	0	00100111	0
linksbündig anordnen	11000000	-4	10011100	-2
Exponenten angleichen	00110000	-2	10011100	-2
Mantissen subtrahieren	10010100	-2		
Mantisse negieren	01101100	-2		

Ergebnis: $108 \times 2^{-2} = 108 \times 1/4 = 27$

(Da bei der Subtraktion ein Überlauf auftritt, ist das Ergebnis negativ: -27)

Tabelle 3. Die Subtraktion und Addition von Fließkommazahlen ist nicht schwierig

```
[WB4]    1: ;  Multiplikationsroutine
[BA5]    2: ;  Geschrieben von Markus Zietlow
[P3Ø]    3: ;
[P74]    4: daten    segment para 'data'
[M35]    5:            ; Definition des Datensegments
[P3Ø]    6: ;
[GW3]    7: faktor1   dw   Ø  ;1. Faktor
[NW3]    8: faktor2   dw   Ø  ;2. Faktor
[GW5]    9: speicher1 dw   Ø  ;1. Zwischenspeicher
[LW5]   1Ø: speicher2 dw   Ø  ;2. Zwischenspeicher
[PW5]   11: speicher3 dw   Ø  ;3. Zwischenspeicher
[TW5]   12: speicher4 dw   Ø  ;4. Zwischenspeicher
[WW5]   13: speicher5 dw   Ø  ;5. Zwischenspeicher
[ZW5]   14: speicher6 dw   Ø  ;6. Zwischenspeicher
[2X5]   15: speicher7 dw   Ø  ;7. Zwischenspeicher
[5X5]   16: speicher8 dw   Ø  ;8. Zwischenspeicher
[P3Ø]   17: ;
[WA6]   18: daten   ends      ; Ende des Datensegments
[P3Ø]   19: ;
[T54]   2Ø: code    segment para 'code'
[P15]   21:           ; Definition des Code-Segments
[6K5]   22: multi32 proc    far  ; Festlegen des
[2R3]   23:                      ; Unterprogrammnamens
[P3Ø]   24: ;
[7Z3]   25:         assume  cs:code, ds:data
[VA3]   26:                      ; Segmente festlegen
[P3Ø]   27: ;
[CC5]   28:    mov  ax,seg data  ; Datensegment
[T15]   29:    mov  ds,ax        ;initialisieren
[3M4]   3Ø:    mov  faktor1,dx   ; Faktoren
[X56]   31:    mov  faktor2,ax   ;zwischenspeichern
[SD5]   32:    mul  bx           ;und multiplizieren
[H86]   33:    mov  speicher1,dx ; 1. Zwischenprodukt
[N15]   34:    mov  speicher2,ax ;abspeichern
[NN2]   35:    mov  ax,faktor1
[8T5]   36:    mul  bx           ; Nächstes Zwischen-
[N16]   37:    mov  speicher3,dx ;produkt bilden und
[ND6]   38:    mov  speicher4,ax ;ebenfalls speichern
[TN2]   39:    mov  ax,faktor2
[4L5]   4Ø:    mul  cx           ; Weiteres Zwischen-
[UØ6]   41:    mov  speicher5,dx ;produkt errechnen
[5V4]   42:    mov  speicher6,ax ;und ablegen
[NN2]   43:    mov  ax,faktor1
[WE5]   44:    mul  cx           ; Letztes Zwischen-
[UF6]   45:    mov  speicher7,dx ;produkt berechnen u.
[256]   46:    mov  speicher8,ax ;im Arbeitsspeicher
[BE1]   47:                      ; sichern
[P3Ø]   48: ;
[686]   49:    mov  ax,speicher2 ; Die untere Hälfte
[UP5]   5Ø:    mov  bx,speicher1 ;des 64-Bit-Ergeb-
[246]   51:    add  bx,speicher4 ;nisses aus dem 1. u.
[E86]   52:    adc  bx,speicher6 ;2.Zwischenspeicher
[V91]   53:                      ;bilden
[Z76]   54:    mov  cx,speicher3 ; gleiche Operation
[R66]   55:    add  cx,speicher5 ;für obere Hälfte
[PL5]   56:    adc  cx,speicher8 ;des Ergebnisses
[ØC5]   57:    mov  dx,speicher7 ;durchführen
[9F6]   58:    adc  dx,Ø  ; Carry-Flag berücksichtigen
[Ø76]   59:    ret       ;Rückkehr ins Hauptprogramm
[P3Ø]   6Ø:    ;
[3B5]   61: multi32 endp   ; Ende der Prozedur
[ZV5]   62: code    ends   ; Ende des Code-Segments
[W45]   63:         end  multi32 ;Ende des Programms
```

Listing 1. Diese Assemblerroutine demonstriert, wie 32-Bit-Werte miteinander multipliziert werden

```
[Z46]    1: ;Divisionsroutine mit Änderung des
[R95]    2: ;  Interrupts Ø (Division durch Ø)
[BA5]    3: ;  Geschrieben von Markus Zietlow
[P3Ø]    4: ;
[DX3]    5: code   segment para 'code'
[V15]    6:          ;Definition des Code-Segments
[7S2]    7: divint proc    far
[KW5]    8:           ;Festlegen des Unterprogrammnamens
[1M5]    9:      assume cs:code  ;Segment festlegen
[P3Ø]   1Ø: ;
[ND1]   11:      cmp   bx,Ø
[N84]   12:                  ;Falls der Divisor gleich Ø
[196]   13:      jnz   weiter  ;ist, dann den Interrupt
[9U4]   14:      int   Ø       ;Ø (Division durch Ø)
[FL1]   15:                    ;aufrufen
[P3Ø]   16: ;
[5T4]   17: weiter: mov di,Ø     ; Den alten
[W75]   18:      mov   es,di     ;Interrupt-Vektor
[V35]   19:      push  es:[di]   ;auf dem Stapel
[334]   2Ø:      push  es:[di+2] ;sichern
[P3Ø]   21: ;
[BZ3]   22:      mov   cx,offset neuintØ
[XM3]   23:                    ;Die Adresse der neuen
[YZ2]   24:      mov   es:[di+2],cx
[PD5]   25:                  ;»Division durch Ø«-Routine
[Y93]   26:      mov   cx,seg neuintØ
[UP3]   27:                    ;im Interruptvektor
[755]   28:      mov   es:[di+2],cx  ; Ø speichern
[WW5]   29:      div   bx      ;Division durchführen
[FØ5]   3Ø:      xor   bx,bx   ;Falls kein Fehler
[4S3]   31:                    ;aufgetreten ist, dann
[4U3]   32:                    ;BX-Register löschen
[127]   33: intback:  pop es:[di+2]  ;Alten Interrupt-
[MD1]   34:                          ;Vektor
[ZV5]   35:      pop   es:[di]       ;wiederherstellen
[4N4]   36:      xor   cx,cx     ;Obere 16 Bit des
[KU3]   37:                      ;Restwertes löschen
[9F4]   38:            ret       ;Rückkehr ins Haupt-
[CL1]   39:                      ;programm
[P3Ø]   4Ø: ;
[GP6]   41: neuintØ: pop  cx    ;Rückkehradresse vom
[DA2]   42:                      ;Stapel holen
[ØZ5]   43:      mov   cx,offset intback ;Neue Rück-
[UC3]   44:                      ;kehradresse auf dem
[ZC1]   45:                      ;Stapel
[493]   46:      push  cx        ;ablegen
[DS4]   47:      push  ax        ;Wenn der Dividend
[VK3]   48:      mov   ax,dx     ;mehr als
[RD4]   49:      xor   dx,dx     ;das 65535fache
[FR5]   5Ø:      div   bx        ;des Divisors beträgt,
[ØD3]   51:      pop   cx        ;dann wird
[724]   52:      push  ax        ;das Ergebnis
[BZ3]   53:      mov   ax,cx     ;auf diesem
[M14]   54:      div   bx        ;einfachen Weg
[BF3]   55:      pop   bx        ;berechnet
[9N4]   56:      iret            ;Rückkehr von der
[X93]   57:                      ;Interrupt-Routine
[P3Ø]   58: ;
[KK5]   59: divint endp          ;Ende der Prozedur
[UY5]   6Ø: code   ends          ;Ende des CodeSegments
[RB5]   61:          end  divint ;Ende des Programms
```

Listing 2. Die Divisionsroutine installiert unter anderem einen neuen Vektor für den Interrupt 0

Wer — wo — wann?

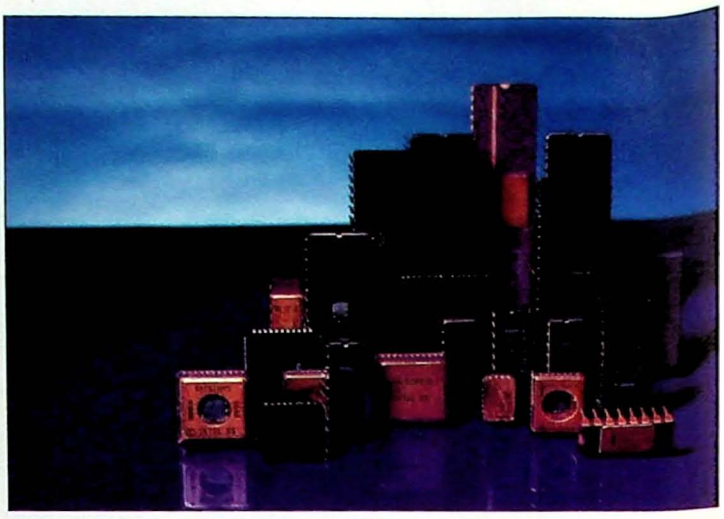
Bei Zuschriften bitten wir, folgendes zu beachten: Haben Sie Fragen, Anregungen oder Programme, die Sie der DOS-Redaktion zukommen lassen möchten, adressieren Sie wie folgt:

Redaktionsbüro Everts & Hagedorn · Redaktion DOS International
Hans-Stießberger-Str. 3 · 8013 Haar/München · Tel. (089) 46 10 01-0

Haben Sie Fragen, die zum Beispiel Abonnements, Kleinanzeigen oder Databox-Bestellungen betreffen, wenden Sie sich bitte an den Verlag:

DMV-Verlag · Postfach 250 · 3440 Eschwege

Assembler am Zeichenbrett



Von der Faszination der Grafik profitieren nicht nur die Zeichenprogramme. Hochsprachen besitzen sogar teilweise eigene Grafik-Grundbefehle. In Assembler dagegen müht sich der Programmierer beim Zeichnen. Allerdings nicht allzusehr, wenn man sich bei der Grafikdarstellung auf die Sonderzeichen des PC-Zeichensatzes beschränkt. In diesem Beitrag findet sogar die Animation in einem Grafikmodus Beachtung.

Bei der Grafikprogrammierung gibt es grundsätzlich zwei Wege, Grafiken beziehungsweise Zeichen darzustellen: erstens mit Hilfe des PC-Zeichensatzes und zweitens durch direkte Ansteuerung des Grafikspeichers.
Den Weg über den PC-Zeichensatz demonstriert Listing 1. Es zeigt, wie sich mit den Grafikzeichen ein Kasten zeichnen und ein einzelnes Sonderzeichen animieren, das heißt in Bewegung versetzen läßt. Dazu greift das Programm über den Video-Interrupt 10hex auf den Bildschirm zu. Dieser Interrupt besteht aus mehreren Funktionen, die in der Tabelle ausführlich beschrieben sind. Der Software-Interrupt mit der Nummer 10hex ist nämlich für den Assemblerprogrammierer das Tor zum Bildschirm. Er gehört zu den vielfältigsten Interrupts des BIOS. Die Funktionen des Video-Interrupts erlauben den Zugriff auf alle Bildschirmkarten, vom Monochromadapter bis hin zur VGA-Karte.
Es existieren zwar auch beim MS-DOS-Interrupt 21hex einige Funktionen für die Bildschirmausgabe, doch eignen sich diese nur für die Textausgabe, weil sie ausschließlich zeichenorientiert arbeiten.
Die Funktionen des Video-Interrupt 10hex sind weitaus leistungsfähiger und vielfältiger. Sie können Bildpunkte setzen und löschen, Bildschirmbereiche aufwärts und abwärts scrollen sowie einzelne Bildschirmpositionen auf Grafikpunkte oder Textzeichen untersuchen.
Für unsere Zwecke sind hauptsächlich die Funktionen 0, 2, 10 und 15 des Video-Interrupts interessant. Die Funktion 15 ermittelt den Videomodus. Sie gibt nach dem Aufruf in Register AL den aktuellen Videomodus, in AH die Anzahl der Zeichen pro Seite und in BH die Nummer der aktuellen Bildschirmseite zurück. Diese Funktion wird am Anfang von Listing 1 benutzt, um die aktuelle Bildschirmseite zu ermitteln, auf die das Programm den Kasten und das animierte Sonderzeichen ausgibt.
Darauf definiert die Funktion 0 den Videomodus. Als Zusatzeffekt wird der gesamte Bildschirm gelöscht. Die Funktion 0 läßt sich auch nur zum Löschen des Bildschirms benutzen, indem zuerst der aktuelle Videomodus über die Funktion 15 eingelesen und darauf über die Funktion 0 wieder ausgegeben wird.
Bevor Listing 1 Zeichen auf den Bildschirm ausgibt, positioniert es mit der Funktion 2 den Cursor, der die Bildschirmposition für die Ausgabe des nächsten Zeichens definiert. Dazu wird im Register DH die Zeile (die Zeile Null ist ganz links), in DL die Spalte (die Spalte Null ist oben) und in BH die Bildschirmseite für die Cursor-Position übergeben.

Durch Löschen eines Zeichens und Vorrücken um eine Position lassen sich Bewegungen simulieren

Die wichtigste Aufgabe in Listing 1 fällt der Funktion 10 zu, denn diese schreibt auf die vorgegebene Bildschirmseite an die aktuelle Cursor-Position ein oder mehrere Zeichen. Die Funktion versetzt den Cursor nicht auf die nächste Bildschirmposition und behält die Farbe der an der Ausgabeposition befindlichen Zeichen bei. Beim Aufruf der Funktion 10 enthält das Register AL den ASCII-Wert des auszugebenden Zeichens und BH die Nummer der Bildschirmseite. Mit CX können Sie zusätzlich bestimmen, wie oft die Funktion das in AL spezifizierte Zeichen hintereinander ausgeben soll.
Nach dem Zeichnen des Kastens folgt die Ausgabe des Sonderzeichens. In Listing 1 wurde das ASCII-Zeichen 1 gewählt, das ein fröhliches Gesicht zeigt. Die Bewegung des Gesichtes kommt dadurch zustande, daß das Sonderzeichen nach einem gewissen Zeitraum durch Überschreiben mit einem Leerzeichen (ASCII-Wert 32) gelöscht und, um eine Position versetzt, erneut ausgegeben wird.
Für die Wartezeit bei der Ausgabe des animierten Sonderzeichens ist diesmal nicht, wie in Listing 1, eine interne Warteschleife, sondern der Systemuhr-Interrupt 1Ahex zuständig. Über diesen Interrupt läßt sich der Zeitzähler des PC direkt ansprechen. Aus diesem Zähler bildet das Betriebssystem beispielsweise die Uhrzeit für den MS-DOS-Befehl »time«.
Während man über die Funktion 2Dhex des MS-DOS-Interrupt 21hex die Uhrzeit des PC in Stunden, Minuten und Sekunden festlegen kann, bietet der Systemuhr-Interrupt 1Ahex weniger Komfort, denn er setzt und liest lediglich einen 32-Bit-Zähler, aus dem der Programmierer selbst die Zeit berechnen muß. Der Zeitzähler arbeitet mit dem Takt des Ticker-Interrupts im PC, was bedeutet, daß der Wert des Zählers 18,2mal pro Sekunde erhöht wird.
Der Systemuhr-Interrupt ändert den Inhalt der Register BX, SI, DI, BP und der Segmentregister nicht. Er besitzt zwei Funktionen, mit denen Sie den Zeitzähler lesen und setzen können. Die Funktion 0 liest den Zeitzähler und gibt im Register AL die Anzahl der vergangenen 24-Stunden-Intervalle, in CX die obere Hälfte und in DX die untere Hälfte des 32-Bit-Zählers zurück. Die Funktion 1 setzt den Zeitzähler, indem der Zählerwert über die Register CX und DX übergeben wird. Beide Funktionen benutzt Listing 1, um eine Wartezeit von jeweils zwei Zählerzyklen, das heißt 2/18,2 Sekunden, zu erzielen.
Durch das Löschen des Zeitzählers in Listing 1 wird allerdings auch die Systemuhr des PC auf 0 Uhr zurückgesetzt. Dieses Problem umgehen Sie dadurch, daß Sie nur zu Beginn den

Zählerstand auslesen und immer mit dem aktuellen Stand vergleichen, bis der Unterschied 2 (oder einen anderen gewünschten Wert) beträgt.
Reichlich verrückt wird die Bildschirmausgabe übrigens, wenn Sie in Listing 1 die Warteschleife weglassen und den Befehl

```
call warten
```

durch den Leerbefehl

```
nop
```

ersetzen.
Bevor Listing 1 das Sonderzeichen ausgibt, überprüft es, ob sich dessen neue Bildschirmposition innerhalb des Kastens befindet. Ist der rechte Rand des Kastens erreicht, wird die x-Koordinate auf den Wert 1 gesetzt und die y-Koordinate inkrementiert. Erreicht das Gesicht schließlich die letzte Spalte in der letzten Zeile des Kastens, so übernimmt das Programm die Ursprungskoordinate als aktuellen Wert.
Die Animation auf der Basis von bildpunktorientierter Grafikausgabe funktioniert auf die gleiche Weise wie das vorher beschriebene Verfahren. Der Unterschied besteht darin, daß die Adressierung der einzelnen Punkte etwas aufwendiger ist. Animierte Figuren erreichen Sie, indem Sie – wie oben – die Figur an der alten Position löschen und an der neuen wieder ausgeben. Listing 2 zeigt Ihnen am Beispiel einer hellblauen Linie, die sich über den Bildschirm bewegt, wie man Grafiken im vierfarbigen 320x200-Punkte-Modus der CGA-, EGA- und VGA-Karten animiert.
Wenn Sie das Beispielprogramm übersetzt haben, werden Sie feststellen, daß die Linie flackert. Um dieses Problem zu vermeiden, behilft man sich mit einem Trick: Die zu animierende Figur wird zuerst an ihrer neuen Position dargestellt und danach von ihrer alten Position gelöscht. Dieses Verfahren stellt Listing 3 als Assemblerprogramm dar. Wenn Sie diese Routine starten, bemerken Sie sofort, daß die Animation wesentlich ruhiger und exakter wirkt.
Mit der Ausnutzung der Assemblersprache haben Sie die schwierigste Hürde, die vor der schnellen und präzisen Grafikprogrammierung steht, überwunden und können endlich die Programmideen verwirklichen, die bislang an der mangelnden Geschwindigkeit der Hochsprachen-Interpreter oder Hochsprachen-Compiler gescheitert sind.

(Markus Zietlow/al)

Die Funktionen des Video-Interrupt 10hex

AH-Register (hex)	Eingabe (Register)	Ausgabe	Bedeutung
00	AL=Videomodus	–	Definiere den Videomodus (Text- oder Grafikdarstellung)
01	CH=000xxxxx (xxxxx=Startrasterzeile von 0 bis 1Fhex) CL=000xxxxx (xxxxx=Endrasterzeile von 0 bis 1Fhex)	–	Definiere Cursor-Größe
02	BH=Bildschirmseite (0 bis 7) (BH=0 für Grafikmodus) DH=Zeile (0 bis 24) DL=Spalte (0 bis 79)	–	Positioniere Cursor (0,0=oben links, 39/79,24=unten rechts)
03	BH=Bildschirmseite (0 bis 7) (BH=0 für Grafikmodus)	CH=Cursor-Startrasterzeile CL=Cursor-Endrasterzeile DH=Zeile DL=Spalte	Lies Cursor-Position (0,0=oben links, 39/79,24=unten rechts)
04	–	AH=0: Lichtgriffel nicht ausgelöst AH=1: Lichtgriffel ausgelöst BX=Grafikspalte (0 bis 639) CH=Grafikzeile (0 bis 199) (CX=Grafikzeile ab EGA-Karte) DH=Zeile des Lichtgriffels (0 bis 24) DL=Spalte des Lichtgriffels (0 bis 79)	Lies Lichtgriffelposition (0,0=oben links, 39/79,24=unten rechts)
05	AL=Bildschirmseite (0 bis 7)	–	Wähle aktive Bildschirmseite
06	AL=Anzahl der Zeilen (frei werdende Zeilen am Fensterende werden gelöscht) (AL=0: Lösche ganzes Fenster) BH=Attribut für die gelöschten Zeilen CH=obere Zeile (0 bis 24) CL=linke Spalte (0 bis 79) DH=untere Zeile (0 bis 24) DL=rechte Spalte (0 bis 79)	–	Scrolle Bildschirm aufwärts
07	AL=Anzahl der Zeilen (frei werdende Zeilen am Fensteranfang werden gelöscht) (AL=0: Lösche ganzes Fenster) BH=Attribut für die gelöschten Zeilen CH=obere Zeile (0 bis 24) CL=linke Spalte (0 bis 79) DH=untere Zeile (0 bis 24) DL=rechte Spalte (0 bis 79)	–	Scrolle Bildschirm abwärts
08	BH=Bildschirmseite (0 bis 7) (BH=0 für Grafikmodus)	AL=ASCII-Zeichen AH=Monochrom- oder Farbattribut	Lies Zeichen und Attribut von der Cursor-Position
09	AL=Zeichen BH=Bildschirmseite (0 bis 7) (BH=0 für Grafikmodus) BL=Monochrom- oder Farbattribut CX=Anzahl der Zeichen	–	Schreibe Zeichen und Attribut auf die Cursor-Position (die Cursor-Position ändert sich nicht)
0A	AL=Zeichen BH=Bildschirmseite (0 bis 7) (BH=0 für Grafikmodus) CX=Anzahl der Zeichen	–	Schreibe Zeichen auf Cursor-Position (die Cursor-Position ändert sich nicht)
0B	Grafikmodus: BH=0: BL enthält Farbe des Bildschirmrahmens und des Hintergrundes (0 bis 15) BH=1: BL enthält Farbpalette BL=0 für Grün, Rot und Braun BL=1 für Kobaltblau, Violett und Weiß Textmodus: BH=0: BL enthält Farbe des Bildschirmrahmens BL=Rahmenfarbe (0 bis 0Fhex: normal, 10 bis 1Fhex: intensiv)	–	Wähle Farbpalette oder Farbe
0C	AL=Farbe (0 bis 15) (Bit 7=1: XOR-Modus) BH=Bildschirmseite (ab EGA-Karte) CX=Spalte (0 bis 639) DX=Zeile (0 bis 349)	–	Setze Grafikpunkt (Pixel)
0D	BH=Bildschirm (ab EGA-Karte) CX=Spalte (0 bis 639) DX=Zeile (0 bis 349)	AL=Farbe des Punktes (0 bis 15)	Lies die Farbe eines Grafikpunktes
0E	AL=ASCII-Zeichen BH=Bildschirmseite (0 bis 7) (BH=0 für Grafikmodus) BL=Vordergrundfarbe (im Grafikmodus)	–	Schreibe Zeichen auf Bildschirm und bewege Cursor vorwärts (Teletype-Emulation)
0F	–	AH=Zeichen pro Zeile AL=Videomodus BH=Bildschirmseite (BH=0: Grafikmodus oder Modus 7)	Ermittle aktuellen Videomodus

AH-Register (hex)	Eingabe (Register)	Ausgabe	Bedeutung
10	AL=Unterfunktionsnummer (hex)	AL=Videomodus AH=Zeichen pro Zeile BH=aktive Bildschirmseite	Setze Register für die Farbpalette (ab EGA-Karte)
	AL=00 BH=Farbwert BL=zu setzendes Paletten-register	(siehe oben)	Setze ein einzelnes Palettenregister
	AL=01 BH=Farbwert	(siehe oben)	Setze Overscan-Register (Border)
	AL=02 ES:DX=Adresse einer 17-Byte-Palettentabelle (Byte 0 bis 15 der Tabelle enthalten die Palettenwerte, Byte 16 enthält den Overscan-Wert)	(siehe oben)	Setze alle Paletten- und Overscan-Register
	AL=03 BL=0: intensive Darstellung BL=1: blinkende Darstellung	(siehe oben)	Wechsle zwischen intensiver und blinkender Darstellung
11	AL=Unterfunktionsnummer (hex)	CX=aktuelle Zeichenhöhe DL=Zeilen pro Bildschirmseite	Charakter-Generator-Routine (ab EGA-Karte)
	AL=00 BH=Anzahl der benötigten Bytes pro Zeichen (0 bis 31) BL=Block für den Zeichensatz (0 bis 1) CX=Anzahl der Zeichen im Zeichensatz (1 bis 256) DX=Zeichensatz-Offset (0 bis 512) ES:BP=Zeiger auf den Anfang des eigenen Zeichensatzes	–	Laden eines eigenen Zeichensatzes und Setzen eines neuen Videomodus
	AL=01 BL=Block für den Zeichensatz (0 bis 1)	–	Laden des ROM-9x14-Monochromzeichensatzes und Setzen eines neuen Videomodus
	AL=02 BL=Block für den Zeichensatz (0 bis 1)	–	Laden des ROM-8x8-Double-Dot-Zeichensatzes und Setzen eines neuen Videomodus
	AL=03 BL=0: Attributbyte wird nicht als Schalter genutzt BL=1: Attributbyte wird als Schalter genutzt	–	Wechsel zwischen zwei Zeichensätzen (Block 0 und 1) und Setzen eines neuen Videomodus
	AL=10 BH=Anzahl der benötigten Bytes pro Zeichen (0 bis 31) BL=Block für den Zeichensatz (0 bis 1) CX=Anzahl der Zeichen im Zeichensatz (1 bis 256) DX=Zeichensatz-Offset (0 bis 512) ES:BP=Zeiger auf den Anfang des eigenen Zeichensatzes	–	Laden eines eigenen Zeichensatzes
	AL=11 BL=Block für den Zeichensatz (0 bis 1)	–	Laden des ROM-9x14-Monochromzeichensatzes
	AL=12 BL=Block für den Zeichensatz (0 bis 1)	–	Laden des ROM-8x8-Double-Dot-Zeichensatzes
	AL=20 ES:BP=Zeiger auf den Anfang der Tabelle mit den Grafikzeichen 128 bis 255	–	Laden der Grafikzeichentabelle
	AL=21 BL=Anzahl der Zeilen einer Bildschirmseite BL=0: Anzahl der Zeilen befindet sich in DL BL=1:14 Zeilen pro Bildschirmseite BL=2:25 Zeilen pro Bildschirmseite BL=3:43 Zeilen pro Bildschirmseite CX=Anzahl der Bytes pro Zeichen	–	Laden eines eigenen Grafikzeichensatzes
11	(DL=Anzahl der Zeilen pro Bildschirmseite, wenn BL=0) ES:BP=Zeiger auf den Anfang des eigenen Zeichensatzes		
	AL=22 BL=Anzahl der Zeilen pro Bildschirmseite (wie unter AL=33 beschrieben)	–	Laden des ROM-8x14-Zeichensatzes
	AL=23 BL=Anzahl der Zeilen pro Bildschirmseite (wie unter AL=33 beschrieben) CX=aktuelle Zeichenhöhe	–	Laden des ROM-8x8-Double-Dot-Zeichensatzes
	AL=30 BL=Zeichensatzzeiger BL=0: INT-1F-Zeiger BL=1: INT-43-Zeiger BL=2: ROM-8x14-Zeiger BL=3: ROM-Double-Dot-Zeiger BL=4: ROM-Double-Dot-Zeiger (Top) BL=5: ROM-9x14-Zeiger (Monochrom)	CX=aktuelle Zeichenhöhe DL=Anzahl der Zeilen pro Bildschirmseite ES:BP=Zeiger (je nach Inhalt von BL vor dem Aufruf)	Ermittle Informationen über einen Zeichensatzzeiger
12	BL=Unterfunktionsnummer (hex) BL=10	– BL=Speichergröße der Grafikkarte BL=0: 64 KByte Bildschirmspeicher BL=1: 128 KByte Bildschirmspeicher BL=10: 192 KByte Bildschirmspeicher BL=11: 256 KByte Bildschirmspeicher BH=Monitortyp BH=0: Farbmonitor BH=1: Monochrommonitor	Ermittle Status oder aktiviere Hardcopy-Routine (ab EGA-Karte) Lies Statusinformationen
	BL=20	–	Aktiviere alternative Hardcopy-Routine für Zeichenhöhen ungleich 8 oder 14
13	AL=Form der Zeichenkette AL=0: Zeichenkette schreiben und Cursor-Position nicht verändern AL=1: Zeichenkette schreiben und Cursor mitführen (Teletype-Emulation) AL=2: Zeichenkette sowie Attributbyte schreiben und Cursor-Position nicht verändern AL=3: Zeichenkette sowie Attributbyte schreiben und Cursor mitführen (Teletype-Emulation) BH=Bildschirmseite BL=Attributbyte (bei AL=0 oder AL=1) CX=Länge der Zeichenkette DH=Zeile der Startposition DL=Spalte der Startposition ES:BX=Adresse der Zeichenkette	–	Gebe eine Zeichenkette auf den Bildschirm aus (ähnlich Funktion 9, aber nur für AT)
14	–	–	Reserviert
15	–	–	Reserviert
16	–	–	Reserviert
17	–	–	Reserviert
18	–	–	Reserviert
19	–	–	Reserviert
1A	AL=Unterfunktionsnummer AL=0	– AL=1A BH=nicht aktiver Monitor BL=aktiver Monitor	Videokonfiguration (VGA-Karte) Ermittle die Videokonfiguration
	AL=1	AL=1A BH=nicht aktiver Monitor BL=aktiver Monitor	Setze die Videokonfiguration
1B	BX=0 ES:DI=Adresse eines 64-Byte-Puffers	AL=1B ES:DI=Puffer, der die gewünschten Daten enthält	Ermittle den Video- und BIOS-Status

AH-Register (hex)	Eingabe (Register)	Ausgabe	Bedeutung
1C	AL = Unterfunktionsnummer	–	Ändere, lies oder setze den Video-Status
	AL = 0: CX = Status Bit 0 = 1: Video-Hardware-Status Bit 1 = 1: Video-BIOS-Datenbereiche Bit 2 = 1: Video-DAC-Status Bit 3 bis 15: Reserviert	AL = 1C BX = Puffergröße in 64-Byte-Blöcken	Lies die Größe des Videopuffers
1C	AL = 1 CX = Status (wie in Unterfunktion 0 beschrieben) ES:BX = Adresse des Puffers	–	Kopiere den Status in den angegebenen Datenblock
	AL = 2 CX = Status (wie in Unterfunktion 0 beschrieben) ES:BX = Adresse des Puffers	–	Setze den Videostatus auf die im Puffer abgespeicherten Werte

Die Text- und Grafikkoordinatenwerte der Funktionen sind in dezimaler Schreibweise angegeben

Tabelle. Die Funktionen des Video-Interrupt 10hex sind zum Teil von der verwendeten Grafikkarte abhängig

```
[6S6]    1: ; Assembler-Programm zur Demonstration
[Y65]    2: ; der Blockgrafik-Eigenschaften
[5E6]    3: ; (c) Redaktionsbüro Everts & Hagedorn
[ERØ]    4: ; 1989
[XT3]    5: code segment para 'code'
[XM2]    6: kastenproc  far
[XE2]    7:       assume cs: code
[883]    8:       push ax       ; Sichern
[3F2]    9:       push bx       ;der
[124]   1Ø:       push cx       ;wichtigsten
[UG4]   11:       push dx       ;Speicherwerte
[F84]   12:       mov  ah,15    ; Ermitteln der
[YS5]   13:       int  1Øh      ;aktiven Bildschirmseite
[P84]   14:       mov  ah,Ø     ; Definieren des
[EE4]   15:       int  1Øh      ;Bildschirmmodus
[W55]   16:       mov  cx,19    ; 19 Zeichen ausgeben
[PF4]   17:       mov  dx,Ø     ;auf Position Ø,Ø
[W55]   18:       mov  ah,2     ; Interruptfunktion
[FS3]   19:                     ;'Positioniere Cursor'
[4T3]   2Ø:       int  1Øh      ;ausführen
[X75]   21:       mov  ah,1Ø    ; Das ASCII-Zeichen 2Ø5
[VW5]   22:       mov  al,2Ø5   ;an der Cursorposition
[GP4]   23:       int  1Øh      ;auf dem Bildschirm
[6H1]   24:                     ;ausgeben
[CM3]   25:       mov  cx,1     ; Einmal das
[9W4]   26:       mov  al,2Ø1   ;ASCII-Zeichen 2Ø1
[9X4]   27:       mov  ah,1Ø    ;auf dem Bildschirm
[P43]   28:       int  1Øh      ;ausgeben
[G24]   29:       add  dl,19    ; Spalte auf die
[HY4]   3Ø:       mov  ah,2     ;rechte, obere Ecke
[1X2]   31:       int  1Øh      ;setzen
[HG2]   32:       mov  cx,1     ;und
[HL4]   33:       mov  al,187   ;entsprechendes
[U63]   34:       mov  ah,1Ø    ;Zeichen
[TK3]   35:       int  1Øh      ;darstellen
[ØH3]   36:       add  dh,9     ; Cursor auf
[1C4]   37:       mov  ah,2     ;untere, rechte
[PK3]   38:       int  1Øh      ;Ecke setzen
[6V4]   39:       mov  al,188   ; ASCII-Zeichen 188
[7H4]   4Ø:       mov  ah,1Ø    ;als Kennzeichen
[6G3]   41:       int  1Øh      ;markieren
[KM4]   42:       sub  dl,19    ; Die untere, linke
[ZK2]   43:       mov  ah,2     ;Ecke
[273]   44:       int  1Øh      ;zeichnen
[GE4]   45:       mov  cx,19    ; Wiederum 19mal
[TA4]   46:       mov  al,2Ø5   ;das Zeichen 2Ø5
[4D4]   47:       mov  ah,1Ø    ;auf den Monitor
[BY2]   48:       int  1Øh      ;bringen
[N13]   49:       mov  cx,1     ; Einmal
[V94]   5Ø:       mov  al,2ØØ   ;das Zeichen 2ØØ
[X94]   51:       int  1Øh      ;positionieren
[7F5]   52:       mov  cx,1     ; Jedes Zeichen einmal
[T12]   53:                     ;darstellen
[5S5]   54:       mov  al,186   ;Zeichen für eine Seite
[KB6]   55: seiten: dec  dh     ; Zeile um 1 vermindern
[TN3]   56:       mov  ah,2     ;und Cursor
[X94]   57:       int  1Øh      ;positionieren
[1A4]   58:       mov  ah,1Ø    ; Danach Zeichen
[P43]   59:       int  1Øh      ;ausgeben
[DG3]   6Ø:       add  dl,19    ; Spalte auf
[6L3]   61:       mov  ah,2     ;die rechte
[YY3]   62:       int  1Øh      ;Seite setzen
[5B4]   63:       mov  ah,1Ø    ; Seitenzeichen
[TK3]   64:       int  1Øh      ;darstellen
[ZF5]   65:       sub  dl,19    ; Spalte auf ursprüng-
[H42]   66:                     ;lichen Wert
[9M5]   67:       cmp  dh,1     ; Erste Zeile erreicht ?
[4Z5]   68:       jne  seiten   ;Wenn nicht, Sprung zu
[UD1]   69:                     ;'seiten'
[GH4]   7Ø:       mov  dx,1Ø1h  ;Koordinaten 1,1
[ER6]   71: animat: mov ah,2    ; Neue Cursorposition
[WH3]   72:       int  1Øh      ;einrichten
[BK3]   73:       mov  al,1     ; Gesicht an
[CM4]   74:       mov  ah,1Ø    ;Cursorposition
[EV2]   75:       int  1Øh      ;zeigen
[2K5]   76:       call warten   ; Aufruf der Zeitver-
[ZY1]   77:                     ;zögerung
[CS3]   78:       mov  ah,11    ; Wenn Taste
[WD4]   79:       int  21h      ;gedrückt, dann
[AT3]   8Ø:       cmp  al,Ø     ;Sprung zum
[A15]   81:       jnz  schluss  ;Ende der Routine
[635]   82:       mov  ah,1Ø    ; Ausgabe anwählen
[893]   83:       mov  al,32    ; Gesicht
[MF4]   84:       int  1Øh      ;wieder löschen
[VC4]   85:       inc  dl       ; Spalte erhöhen
[ZZ4]   86:       cmp  dl,19    ; Wenn rechter Rand
[7R2]   87:                     :nicht erreicht,
[ØS5]   88:       jne  animat   ;Routine wiederholen,
[7K4]   89:       mov  dl,1     ;sonst Spalte = 1
[474]   9Ø:       inc  dh       ; Zeile erhöhen
[VV4]   91:       cmp  dh,9     ; Wenn untere Zeile
[9R2]   92:                     ;nicht erreicht,
[ØS5]   93:       jne  animat   ;Routine wiederholen,
[XC4]   94:       mov  dh,1     ;sonst Zeile = 1
[135]   95:       jmp  animat   ;und weitermachen
[P3Ø]   96: ;
[DW5]   97: warten proc         ; Anfang der Zeitver-
[HG3]   98:                     ;zögerungsroutine
[UL5]   99:       push cx       ; Anderweitig benutzte
[7T4]  1ØØ:       push dx       ;Register sichern
[8M5]  1Ø1:       xor  cx,cx    ; CX-Register löschen
[EM5]  1Ø2:       xor  dx,dx    ; DX-Register löschen
[6D4]  1Ø3:       mov  ah,1     ; Systemzeit auf
[7R3]  1Ø4:       int  1ah      ;Null setzen
[WZ4]  1Ø5:       xor  ah,ah    ; Funktion Ø liest
[1H2]  1Ø6:                     ;die Systemuhr
[AD5]  1Ø7: zehntel: int 1ah      ; Uhr auslesen
[L84]  1Ø8:       cmp  dl,2     ; 2/18,2 Sekunden
[4F3]  1Ø9:       jne  zehntel  ;warten
[PA5]  11Ø:       pop  dx       ; Gesicherte Register
[HA5]  111:       pop  cx       ;wieder zurückholen
[MF5]  112:       ret           ; Rückkehr z. Hauptprog.
[8M5]  113: warten endp         ; Ende der Prozedur
[P3Ø]  114: ;
[8K5]  115: schluss: mov dh,15 ; Cursor unter
[3L3]  116:       mov  ah,2     ;dem Kasten
[X94]  117:       int  1Øh      ;positionieren
[1K3]  118:       pop  dx       ; Die auf dem
[R25]  119:       pop  cx       ;Stapel gesicherten
[TZ2]  12Ø:       pop  bx       ;Werte
[3V4]  121:       pop  ax       ;wiederherstellen
[1V4]  122:       mov  ah,4ch   ; Rückkehr in die
[WS3]  123:       int  21h      ;MS-DOS-Ebene
[ML5]  124: kasten endp         ; Ende der Prozedur
[3X5]  125: code ends           ; Ende d. Code-Segments
[HB5]  126:       end   kasten  ; Ende des Programms
```

Listing 1. Für den Aufbau von Bildern und sogar für Animationen eignen sich die Sonderzeichen des PC-Zeichensatzes

```
[9P6]    1: ; Beispielprogramm, das eine Linie über
[BG4]    2: ;          den Bildschirm bewegt
[W44]    3: ; Autor: Markus Zietlow
[5E6]    4: ; (c) Redaktionsbüro Everts & Hagedorn
[YG5]    5: voffset equ  Ø       ; Byte-Offset der
[4P2]    6:                      ;Koordinate (Ø;Ø)
[925]    7: videosg equ  Øb8ØØh ; Anfang des
[ØL3]    8:                      ;Bildschirmspeichers
[WU5]    9: video   equ  1Øh     ; Adresse des Video-
[272]   1Ø:                      ;Interrupts
[VA5]   11: code  segment       ; Programmanfang
[HH5]   12:       org  Ø1ØØh    ; Startadresse für COM-
[ZZØ]   13:                     ;Datei
[H65]   14: assume cs:code, ds:code, es:code
```

Listing 2. Die Grafikanimation ist wesentlich aufwendiger als die Animation über Sonderzeichen (ein senkrechter blauer Strich wandert von links über den Bildschirm)

```
[723]   15: beispiel proc near
[H27]   16: start: call lsmode ; Bildschirmmodus lesen
[FE4]   17:        push ax     ; Modus sichern
[WX5]   18:        mov  ax,4h  ; Grafikmodus aktivieren
[666]   19:        call srmode ;mit dem Video-Interrupt
[WV6]   20:   ;Hellblaue Linie über Bildschirm bewegen
[YR4]   21:        xor  ax,ax  ; Alle angegebenen
[EK3]   22:        xor  bx,bx  ;Register
[TM3]   23:        xor  cx,cx  ;löschen
[TK6]   24: start1: inc bx     ; X-Koordinate erhöhen
[AY4]   25:        mov  ch, 1  ; Farbattr. hellblau
[N54]   26:        call linie  ; Linie ziehen
[ZE5]   27:        mov  ch, 0  ; Farbattribut schwarz
[N54]   28:        call linie  ; Linie ziehen
[0Y4]   29:        cmp  bx,319 ; Wenn rechter Rand
[EK5]   30:        jnz  start1 ;noch nicht erreicht
[XS1]   31:                    ;--> start1
[CN5]   32:        call taste  ; Tastendruck abwarten
[5B4]   33:        pop  ax     ; Original-Modus
[R92]   34:                    ;wieder lesen
[676]   35:        call srmode ; Bildschirmmodus setzen
[W66]   36:        mov  ah,4ch ; Progr. über Interrupt
[FA3]   37:        int  21h    ;21h beenden
[W16]   38: linie: push ax     ; X- und Y-Koordinate
[463]   39:        push bx     ;sichern
[HG1]   40: schleife:
[V15]   41:        push ax     ; AX- und BX-Register
[463]   42:        push bx     ;sichern
[745]   43:        call pixadr ; Bitmaske und Byte-
[G23]   44:                    ;adresse berechnen
[8M5]   45:        mov  al, ch ; Farbnummer festlegen
[UZ5]   46:        shl  ax, cl ; Bitmaske aktualisieren
[0D6]   47: Pixel: not  ah     ; Bitmaske invertieren
[VA6]   48:        and  es:[bx],ah ; Bildpunkt löschen
[356]   49:        or   es:[bx],al ; Bildpunkt ersetzen
[966]   50: weiter: pop bx     ; X- und Y-Koordinate
[US4]   51:        pop  ax     ;wiederherstellen
[SB5]   52:        inc  ax     ; Y-Koord. um 1 erhöhen
[WV5]   53:        cmp  ax,199 ; Vergleich mit 199:wenn
[UY3]   54:                    ;unterer Rand nicht er-
[NG5]   55:        jnz  schleife ;reicht ==> schleife
[BU5]   56:        pop  bx     ; Ursprüngliche Koordi-
[CR5]   57:        pop  ax     ;naten wiederherstellen
[556]   58:        ret    ; Rückkehr ins Hauptprogramm
[U64]   59:  ;     Bildschirmmodus lesen
[L06]   60: lsmode: mov ah,0fh ;Aktuellen Video-
[AR4]   61:        int  video  ;Modus ermitteln
[3P0]   62:        ret
[N54]   63: ; Bildschirmmodus setzen
[TY5]   64: srmode: mov ah,00h ; Den in AL gespei-
[6C5]   65:        int  video  ;cherten Video-Modus
[132]   66:                    ;aktivieren
[3P0]   67:        ret
[YG6]   68: ; Bytenummer und Bitmaske eines Pixels
[SK6]   69: ; berechnen: das bedeutet, Pufferadresse
[3F6]   70: ; eines Bildpunkts im 320x200-Modus mit
[M63]   71: ; 4 Farben bestimmen
[W93]   72: pixadr: mov  cl, bl
[073]   73:        xchg ah,al  ; AX=256*y
[GZ4]   74:        shr  ax,1   ;und durch 2 geteilt
[PH1]   75:        add  bh,al
[B14]   76:        xor  al,al  ; AL löschen
[873]   77:        add  bx,ax  ; BX=AX+x
[075]   78:        shr  ax,1   ; AX erhält den Wert
[VC2]   79:        shr  ax,1   ;64*y
[YE3]   80:        add  bx,ax  ; BX=320*y+x
[775]   81:        shr  bx,1   ; BX erhält den Wert
[PH4]   82:        shr  bx,1   ;X geteilt durch 8
[0P5]   83:        add  bx,voffset ; BX=Anfangsadres-
[G13]   84:                    ;se im Videopuffer
[1P5]   85:        mov  ax,videosg ; ES enthält Seg-
[052]   86:                    ;mentadresse
[MZ4]   87:        mov  es,ax  ;des Video-Puffers
[7H1]   88:        mov  ah, 3
[VK1]   89:        and  cl,ah
[4P1]   90:        xor  cl,ah
[6G1]   91:        shl  cl, 1
[3P0]   92:        ret
[R25]   93:        ; Warten auf einen Tastendruck
[115]   94: taste: mov  ah,0 ; Ein Zeichen von
[E34]   95:        int  16h  ;Tastatur lesen
[3P0]   96:        ret
[0M5]   97: beispiel endp   ; Ende der Prozedur
[065]   98: code  ends      ; Ende des Programms
[KA5]   99:        end      ; Festlegen Startadresse
[WX4]  100:        end      ; Ende des Quelltextes
```

Listing 2. Die Grafikanimation ist wesentlich aufwendiger als die Animation über Sonderzeichen (ein senkrechter blauer Strich wandert von links über den Bildschirm)

```
[9P6]    1: ; Beispielprogramm, das eine Linie über
[3G5]    2: ;  den Bildschirm bewegt (Version 2)
[W44]    3: ; Autor: Markus Zietlow
[5E6]    4: ; (c) Redaktionsbüro Everts & Hagedorn
[YG5]    5: voffset equ  0       ; Byte-Offset der
[4P2]    6:                      ;Koordinate (0;0)
[EY5]    7: videosg equ  0b800h ; Anfang des Bild-
[2X2]    8:                      ;schirmspeichers
[WU5]    9: video   equ  10h     ; Adresse des Video-
[272]   10:                      ;Interrupts
[PE5]   11: code segment         ; Programmanfang
[BY4]   12:       org  0100h     ; Startadresse für
[9G2]   13:                      ;eine COM-Datei
[H65]   14: assume cs:code, ds:code, es:code
[723]   15: beispiel proc near
[H27]   16: start: call lsmode ; Bildschirmmodus lesen
[FE4]   17:       push ax      ; Modus sichern
[WX5]   18:       mov  ax,4h   ; Grafikmodus aktivieren
[666]   19:       call srmode ;mit dem Video-Interrupt
[986]   20: ; Hellblaue Linie über den Bildschirm
[M43]   21: ; bewegen (Version 2)
[K25]   22:       xor  ax,ax   ; Alle angegebenen
[PT3]   23:       xor  bx,bx   ;Register
[LM3]   24:       xor  cx,cx   ;löschen
[2N6]   25: start1: inc  bx    ; X-Koordinate erhöhen
[XU5]   26:       mov  ch, 1   ; Farbattribut hellblau
[VA4]   27:       call linie   ; Linie ziehen
[844]   28:       dec  bx      ; X-Koordinate
[742]   29:                    ;vermindern
[9S5]   30:       mov  ch, 0   ; Farbattribut schwarz
[VA4]   31:       call linie   ; Linie ziehen
[GZ5]   32:       inc  bx      ; Ursprüngliche X-Koor-
[DL3]   33:                    ;dinate restaurieren
[U45]   34:       cmp  bx,319  ; Wenn rechter Rand
[ML5]   35:       jnz  start1  ;noch nicht erreicht
[XS1]   36:                    ;--> start1
[2X5]   37:       call taste   ; Tastendruck abwarten
[WN5]   38:       pop  ax      ; Original-Modus wieder
[N11]   39:                    ;lesen
[K76]   40:       call srmode  ;Bildschirmmodus setzen
[916]   41:       mov  ah,4ch  ; Programm über Inter-
[HL4]   42:       int  21h     ;rupt 21h beenden
[P30]   43: ;
[T66]   44: linie: push ax     ; X- und Y-Koordinate
[693]   45:       push bx      ;sichern
[HG1]   46: schleife:
[BF5]   47:       push ax      ; AX- und BX-Register
[693]   48:       push bx      ;sichern
[5K5]   49:       call pixadr  ; Bitmaske und Byte-
[G23]   50:                    ;adresse berechnen
[EV5]   51:       mov  al, ch  ; Farbnummer festlegen
[H56]   52:       shl  ax, cl  ;Bitmaske aktualisieren
[K96]   53: Pixel: not  ah     ; Bitmaske invertieren
[VA6]   54:       and  es:[bx],ah ; Bildpunkt löschen
[356]   55:       or   es:[bx],al ; Bildpunkt ersetzen
[E56]   56: weiter: pop  bx    ; X- u. Y-Koordinate
[CY4]   57:       pop  ax      ;wiederherstellen
[1X4]   58:       inc  ax      ; Y-Koordinate um 1
[1K1]   59:                    ;erhöhen
[345]   60:       cmp  ax,199  ; Vergleich mit 199:
[M33]   61:                    ;wenn unterer Rand
[DU5]   62:       jnz  schleife ;nicht erreicht, dann
[RY1]   63:                    ;--> schleife
[K85]   64:       pop  bx      ; Ursprüngliche Ko-
[7Y4]   65:       pop  ax      ;ordinaten wieder-
[N22]   66:                    ;herstellen
[0X5]   67:       ret    ;Rückkehr ins Hauptprogramm
[E14]   68:       ; Bildschirmmodus lesen
[PG6]   69: lsmode: mov ah,0fh ; Den aktuellen Video-
[0Z4]   70:       int  video   ;Modus ermitteln
[3P0]   71:       ret
[N54]   72: ; Bildschirmmodus setzen
[MG6]   73: srmode: mov ah,00h ; In AL gespeicherten
[B06]   74:       int  video   ;Video-Modus aktivieren
[3P0]   75:       ret
[HZ6]   76: ; Bytenummer und Bitmaske eines Pixels be-
[D66]   77: ;rechnen: das bedeutet, Pufferadresse
[R66]   78: ;eines Bildpunkts im 320x200-Modus mit 4
[AY2]   79: ;Farben bestimmen
[NW2]   80: pixadr: mov cl,bl
[YC3]   81:       xchg ah,al   ; AX=256*y
[CZ4]   82:       shr  ax,1    ;und durch 2 geteilt
[PH1]   83:       add  bh,al
[V14]   84:       xor  al,al   ; AL löschen
[SH3]   85:       add  bx,ax   ; BX=AX+x
[CB5]   86:       shr  ax,1    ; AX erhält den Wert
[3R2]   87:       shr  ax,1    ;64*y
[CR3]   88:       add  bx,ax   ; BX=320*y+x
[HB5]   89:       shr  bx,1    ; BX erhält den Wert
[CR4]   90:       shr  bx,1    ;X geteilt durch 8
[626]   91:       add  bx,voffset ; BX=Anfangsadresse
[ZR2]   92:                    ;im Video-Puffer
[PD6]   93:       mov  ax,videosg ; ES enthält Segment-
[F05]   94:       mov  es,ax   ;adresse des Video-
[PF1]   95:                    ;Puffers
[7H1]   96:       mov  ah, 3
[VK1]   97:       and  cl,ah
[4P1]   98:       xor  cl,ah
[6G1]   99:       shl  cl, 1
[3P0]  100:       ret
[R25]  101: ; Warten auf einen Tastendruck
[115]  102: taste: mov  ah,0 ; Ein Zeichen von
[CX3]  103:       int  16h ;Tastatur lesen
[3P0]  104:       ret
[AE5]  105: beispiel endp ; Ende der Prozedur
[G85]  106: code ends     ; Ende des Programms
[965]  107:       end      ; Festlegen Startadresse
[XX4]  108:       end      ; Ende des Quelltextes
```

Listing 3. Mit der zweiten Methode wirkt die animierte Grafik wesentlich ruhiger

C-SHOP

Christel's - Software - Shop

Lindlarer Str. 141 * 5063 Overath-Immekeppel

TELEFON : 02204 - 74075
FAX : 02207 - 4886

IBM - SOFTWARE

CLIPPER (S87)	D	1.489,00 DM
DBASE IV	D	1.478,00 DM
DBASE IV ENTWICKLER PACK	D	2.299,00 DM
F & A 3.0	D	978,00 DM
FRAMEWORK III	D	1.298,00 DM
GEM 1ST WORD PLUS	D	407,00 DM
GEM ARTLINE	D	979,00 DM
GEM DESKTOP PUBLISHER	D	833,00 DM
GEM DRAW PLUS	D	519,00 DM
GEM GRAPH	D	499,00 DM
HARVARD GRAPHICS	D	979,00 DM
LOTUS 1-2-3	D	866,00 DM
LOTUS SYMPHONY	D	1.299,00 DM
MS-CHART 3.0	D	722,00 DM
MS-EXCEL	D	999,00 DM
MS-MULTIPLAN 4.0	D	528,00 DM
MS-QUICK-C	D	208,00 DM
MS-QUICKBASIC COMPILER 4.5	D	208,00 DM
MS-WINDOWS-286 2.1	D	288,00 DM
MS-WINDOWS-386 2.1	D	449,00 DM
MS-WORD 4.0	D	895,00 DM
MS-WORKS	D	378,00 DM
NORTON COMMANDER 2.0	D	188,00 DM
NORTON UTILITIES ADV 4.5	D	244,00 DM
OPEN ACCESS II	D	1.389,00 DM
PAGEMAKER 3.0	D	1.459,00 DM
PC TOOLS DELUXE 4.3	D	118,00 DM
PC TOLLS DELUXE 5.1	E/D	149,00 DM
SIDEKICK PLUS	D	339,00 DM
TURBO ASSEMBLER	D	299,00 DM
TURBO BASIC	D	198,00 DM
TURBO C 2.0	D	268,00 DM
TURBO C PROFESSIONAL	D	448,00 DM
TURBO PASCAL 5.0	D	268,00 DM
TURBO PASCAL PROFESSIONAL	D	448,00 DM
TURBO PROFI PROFI	D	568,00 DM
VENTURA PUBLISHER 2.0	D	1.898,00 DM
WORDPERFECT 5.0	D	978,00 DM
WORDSTAR 5.0	D	768,00 DM
XTREE	D	199,00 DM
XTREE PRO	D	278,00 DM

MONITORE

EIZO FLEXSCAN 9060S 14"	1.699,00 DM
EIZO FLEXSCAN 9070S 16"	2.099,00 DM
EIZO FLEXSCAN 9500 20"	5.999,00 DM
EIZO 6500 SET 21" WEIa	5.599,00 DM
EIZO 9060S + EIZO MD-B07	2.269,00 DM
EIZO 9070S + EIZO MD-B10	2.979,00 DM
NEC MULTISYNC II 14"	1.299,00 DM
NEC MULTISYNC XL 20"	4.599,00 DM
VISA MC-8535 19" FARBMONITOR	2.998,00 DM
VISA MC-6700 MULTISYNC 14"	1.098,00 DM
VISA FM 1420 14"	298,00 DM
MONITOR 14" WEIa/AMBER	239,00 DM

GRAPHIC - KARTEN

EIZO MD-B07 VGA-KARTE	649,00 DM
EIZO MD-B10 VGA 16 BIT 512 KB	899,00 DM
KNC-SUPER VGA PLUS 16 BIT	998,00 DM
ORCHID PRO DESIGNER + 16 BIT.	949,00 DM
MAX VGA KARTE 800 x 600	499,00 DM
MAX EGA KARTE 800 x 600	399,00 DM
ULTIMATE EGA-1024	448,00 DM

FESTPLATTEN

SEAGATE ST 125 20 MB	529,00 DM
SEAGATE ST 125-1 20 MB	599,00 DM
SEAGATE ST 138R 30 MB	599,00 DM
SEAGATE ST 157R 49 MB	889,00 DM
SEAGATE ST 225 20 MB	459,00 DM
SEAGATE ST 238R 30 MB	489,00 DM
SEAGATE ST 251 40 MB	779,00 DM
SEAGATE ST 251-1 40 MB	879,00 DM
SEAGATE ST 4096 80 MB	1.299,00 DM
MINISCRIBE 8425F 20 MB	498,00 DM
MINISCRIBE 3053 44 MB	898,00 DM
MINISCRIBE 3085 71 MB	1.498,00 DM
SEAGATE DRIVE CARD 20 MB	679,00 DM
SEAGATE DRIVE CARD 30 MB	779,00 DM
SEAGATE DRIVE CARD 50 MB	1.169,00 DM

CONTROLLER

WD 1002 AWX-1 XT 2 HD	129,00 DM
WD 1002 A27-X XT 2 HD RLL	139,00 DM
WD 1003 WA2 AT 2 HD 2 FDD	259,00 DM
WD 1003 WAH AT 2 HD	229,00 DM
WD 1006 VMM-2 AT 2 HD 2 FDD	299,00 DM
WD 1006 VMM-1 AT 2 HD	279,00 DM
WD 1006 VMR-2 AT 2HD 2FDD RLL	369,00 DM
NCL 5525 AT 2HD 2FDD MFM	269,00 DM
NCL 5355 AT 2HD 2FDD ESDI	369,00 DM
NCL 5235 AT 2HD 2FDD RLL	389,00 DM
OMTI 8629 AT 2HD 2FDD ESDI	459,00 DM

NOVELL - NETWARE

ADVANCED NETWARE 86	3.888,00 DM
ELS - 1 286 4 USER	1.539,00 DM
ELS - 2 286 8 USER	3.239,00 DM
ADVANCED NETWARE 286 STF	6.399,00 DM

SONSTIGES

GENIUS GM 6 PLUS			82,00 DM
GENIUS GM 6000			92,00 DM
DFI HS 3000 HANDY SCANNER			450,00 DM
NCE MOUSE 869 PC			99,00 DM
LOGITECH MOUSE C7			189,00 DM
NO NAME 5,25"	2D	100 St.	72,00 DM
NO NAME 5,25"	2HD	100 St.	189,00 DM
NO NAME 3,5"	2DD	100 St.	259,00 DM
MAXELL MF 3,5"	2DD	100 St.	299,00 DM

BTX TERM PC ..... 209,00 DM
DFI MD 2400 MODEMSTECKKARTE .... 348,00 DM

Betrieb am Postnetz der Bundespost verboten

PAGEMAKER 3.0	1.459,00 DM
dBase IV	1.478,00 DM
F & A	978,00 DM
Clipper S87	1.489,00 DM

LASERDRUCKER

KYOCERA F-1000	5.178,00 DM
KYOCERA F-1200	7.678,00 DM
KYOCERA F-2200	9.998,00 DM
KYOCERA F-3000	12.978,00 DM
KYOCERA P-2000 POSTSCRIPT	12.999,00 DM
SHARP JX - 9300	2.959,00 DM
EPSON GQ-3500	4.399,00 DM
CANON LBP 8-II	4.498,00 DM
HEWLETT PACKARD DESKJET	1.798,00 DM

NEC DRUCKER

12 MONATE GARANTIE

NEC P2200	799,00 DM
NEC P6 PLUS	1.469,00 DM
NEC P7 PLUS	1.799,00 DM

EPSON DRUCKER

EPSON LQ-400	779,00 DM
EPSON LQ-550	948,00 DM
EPSON LQ-850	1.398,00 DM
EPSON LQ-1050	1.798,00 DM
EPSON LQ-2550	3.299,00 DM

STAR DRUCKER

STAR LC-10	498,00 DM
STAR LC 24-10	798,00 DM

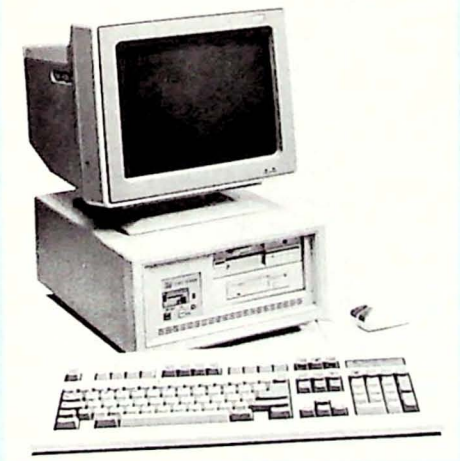
PANASONIC DRUCKER

PANASONIC KX-P 1124	999,00 DM

OKIDATA DRUCKER

OKI ML 390	1.369,00 DM
OKI ML 391	1.769,00 DM

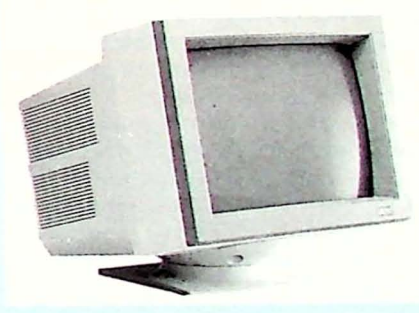
BUS-AT, Desk 286 N



mit Monitor Visa 1420 14" und 20 MB Festplatte

3.333,00 DM

VGA-MONITOR



899,00 DM

NEC MULTISYNC II	1.299,00 DM
NEC P2200	799,00 DM
NEC P6 PLUS	1.469,00 DM

COMPAQ	TANDON	COMMODORE
NEC	EPSON	STAR
SHARP	BROTHER	KEYOCERA

Fragen Sie uns nach den Aktuellen Preisen
Firmen und Behoerden werden auf Rechnung beliefert.

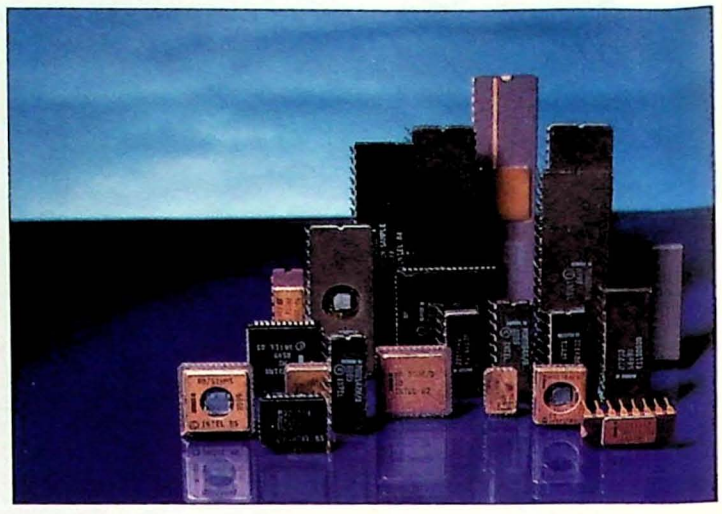
KUNDENBERATUNG - KUNDENSERVICE - KUNDENBETREUUNG werden bei uns GROSS geschrieben.

! HÄNDLERANFRAGEN ERWÜNSCHT ! Mo. - Fr. von 9.00 - 18.00

Unsere Produkte können Sie auch bei unseren Partnern Abholen:

CPS	* Mundsburger Damm 30	* 2000 Hamburg 76	* Tel.: 040 - 222388
CPS	* Herderstraße 32	* 4000 Düsseldorf 1	* Tel.: 0211 - 678674
TECO	* Gummersbacher Str. 9a	* 5270 Gummersbach	* Tel.: 02261 - 21029

Musikalische Maschinensprache



Natürlich macht die Maschinensprache aus Ihrem Computer kein Klavier. Aber nichtsdestotrotz kann der PC ein bißchen mehr, als nur Piepser von sich zu geben. Zum Erzeugen von Tönen beschränkt sich dieser Beitrag allerdings auf die Programmierung der Register des Timer-Bausteins 8253.

Viele Programmiersprachen enthalten in ihrem Befehlssatz Kommandos, mit denen sich über den PC-Lautsprecher Töne ausgeben lassen. Bei der Assemblersprache ist dies jedoch nicht der Fall, so daß Sie sich selbst Routinen zur Tonausgabe schreiben müssen. Hat man aber einmal verstanden, wie ein Ton prinzipiell erzeugt wird, ist dies nicht weiter schwer.

Töne (und auch Geräusche) sind nichts anderes als Schwingungen in der Luft, das heißt rasch wechselnde Druckunterschiede, die beispielsweise dann entstehen, wenn sich die Membrane eines Lautsprechers vor- und zurückbewegt. Wenn mehrere Schwingungen in kurzen Zeitabständen aufeinanderfolgen, entstehen bei diesem Vorgang Töne.

Die Anzahl der Schwingungen innerhalb einer bestimmten Zeitspanne wird als Frequenz bezeichnet und in der Einheit »Hertz« (Schwingungen pro Sekunde) gemessen. Je mehr Schwingungen ein Lautsprecher in einem festen Zeitraum erzeugt, desto höher ist der dabei entstehende Ton und die Frequenz.

Man unterscheidet zwischen akustischen und musikalischen Tönen. Während akustische Töne beliebige Frequenzen besitzen können, sind die Frequenzen für die Töne einer musikalischen Oktave genau festgelegt.

Die Tabelle zeigt die Frequenzen für die Töne der acht unteren Oktaven. Unterhalb der Oktave, die mit der Frequenz von 16,35 Hz beginnt, existiert keine weitere Oktave, weil der Mensch erst Schwingungen ab einer Frequenz von 16 Hz als Töne wahrnimmt. Alle Frequenzen unterhalb dieses Wertes werden nur als Brummen oder Dröhnen empfunden.

Der Bezugspunkt für alle Töne in der Tabelle ist der Ton a in der fünften Oktave mit einer Frequenz von exakt 440 Hz. Dieser Ton wird auch als Kammerton a bezeichnet. Weil die Frequenz des Tons a in der benachbarten Oktave stets doppelt oder halb so groß ist wie die ursprüngliche Frequenz und eine Oktave aus 12 Halbtönen besteht, ist der Unterschied zwischen zwei benachbarten Halbtönen stets ein Faktor mit dem Wert der zwölften Wurzel aus 2. Das entspricht in etwa dem Wert 1,0594631.

Um nun, von der Frequenz eines Halbtons ausgehend, die Frequenz des unmittelbar folgenden Halbtons zu berechnen, müssen Sie nur die Frequenz des ersten Halbtons mit 1,0594631 multiplizieren. Das ist das ganze Geheimnis der Tabelle!

Der Lautsprecher wird über den Timer-Baustein 8253 und den Peripheriebaustein 8255 angesteuert

Doch wie lassen sich jetzt mit den Informationen aus der Tabelle Töne erzeugen? Wenn der PC-Lautsprecher beispielsweise den Kammerton a erzeugen soll, müssen Sie den Timer-Baustein 8253, der im PC die Impulse für den Lautsprecher liefert, so programmieren, daß er 440mal in der Sekunde ein elektrisches Signal ausgibt, das die Lautsprechermembrane zum Schwingen anregt. Zu diesem Zweck müssen Sie in den 8253 über die Portadresse 42hex einen Wert schreiben, der die gewünschte Frequenz angibt. Doch im PC ist nicht nur der 8253 für die Ansteuerung des Lautsprechers zuständig. Fast alle Ein- und Ausgabeoperationen steuert der programmierbare Peripheriebaustein 8255, der über drei parallele 8-Bit-Schnittstellen verfügt. Zwei dieser Schnittstellen sind für die Eingabe und eine ist für die Ausgabe von Daten zuständig.

Alle drei Schnittstellen des 8255 sind mit 8-Bit-Registern gepuffert. Die Register für die beiden Eingabeschnittstellen besitzen die Portadresse 60hex und 62hex, das Register für die Ausgabeschnittstelle wird über die Portadresse 61hex angesprochen. Die unteren beiden Bits des letzteren Registers steuern den PC-Lautsprecher. Ist das Bit 0 dieses Registers auf 1 gesetzt, so wird der zweite Kanal des Timer-Bausteins 8253 dazu veranlaßt, ein Rechtecksignal zu erzeugen, dessen Frequenz Sie über die 8253-Register definieren können. Ist auch Bit 1 des Registers unter der Portadresse 61hex auf 1 gesetzt, so wird das Rechtecksignal des 8253 an den Lautsprecher geführt, und dessen Membrane erzeugt einen Ton. Die Frequenz dieses Tons entspricht der Frequenz des Rechtecksignals. Sobald Bit 1 auf 0 zurückgesetzt wird, ist auch das Rechtecksignal des 8253 wieder vom Lautsprecher getrennt, und dieser verstummt augenbicklich.

Die wichtigsten Tonfrequenzen auf einen Blick

Ton	Frequenz Oktave 1	Frequenz Oktave 2	Frequenz Oktave 3	Frequenz Oktave 4	Frequenz Oktave 5	Frequenz Oktave 6	Frequenz Oktave 7	Frequenz Oktave 8
c	16,35	32,70	65,41	130,81	261,63	523,25	1046,50	2093,00
cis	17,32	34,65	69,30	138,59	277,18	554,37	1108,74	2217,46
d	18,35	36,71	73,42	146,83	293,66	587,33	1174,66	2349,32
dis	19,45	38,89	77,78	155,56	311,13	622,25	1244,51	2489,02
e	20,60	41,20	82,41	164,81	329,63	659,26	1328,51	2637,02
f	21,83	43,65	87,31	174,61	349,23	698,46	1396,91	2793,83
fis	23,12	46,25	92,50	185,00	369,99	739,99	1479,98	2959,96
g	24,50	49,00	98,00	196,00	392,00	783,99	1567,98	3135,96
gis	25,96	51,91	103,83	207,65	415,30	830,61	1661,22	3322,44
a	27,50	55,00	110,00	220,00	440,00	880,00	1760,00	3520,00
ais	29,14	58,27	116,54	233,08	466,16	923,33	1864,66	3729,31
h	30,87	61,74	123,47	246,94	493,88	987,77	1975,53	3951,07

Tabelle. Die Töne der unteren acht Oktaven mit den zugehörigen Frequenzen

Der Ihnen vertrauteste Ton Ihres PC ist wahrscheinlich das Piepsen, das nach dem Einschalten und dem folgenden Selbsttest des Computers ertönt. Auch dieses Piepsen wird über den Timer-Baustein 8253 und den Peripheriebaustein 8255 gesteuert.

Das Listing zeigt nun konkret, wie Sie mit dem PC-Lautsprecher Töne erzeugen und sogar eine ganze Melodie spielen können. Damit nicht die Tonlänge und Tonhöhe von der Taktfrequenz des verwendeten Computers abhängt und störende Interrupts die Noten der Melodie verzerren, wird der Lautsprecher nicht direkt, sondern wie bereits beschrieben über den Timer-Baustein 8253 angesteuert. Der Timer-Baustein erzeugt das Rechtecksignal für den Lautsprecher, ohne daß sich der Prozessor des PC darum kümmern muß. Aus diesem Grund kann auch der weitere Programmablauf über den Prozessor die Tonerzeugung nicht beeinflussen.

Um dem 8253 einen bestimmten Ton zu entlocken, müssen Sie in das Register mit der Portadresse 42hex einen 16-Bit-Wert schreiben, durch den die interne Taktfrequenz des 8253 geteilt wird. Das Ergebnis dieser Division gibt die Frequenz für das zu erzeugende Rechtecksignal an, das die Lautsprechermembrane zum Schwingen bringt.

Vor der Übergabe des Frequenzteilers müssen Sie jedoch noch das Steuerregister des 8253 über die Portadresse 43hex mit dem Wert 182 laden, um den Baustein für den Lautsprecherbetrieb zu initialisieren.

Die interne Taktfrequenz des 8253 beträgt 1193180 Hz. Das ist ein Zwölftel des PC-Systemtaktes von 14,31818 MHz, aus dem beispielsweise auch der 4,772727-MHz-Takt für den Prozessor gewonnen wird. Auch in PCs, die mit höheren Taktfrequenzen arbeiten, verwendet der 8253 intern einen Takt von 1193180 Hz, so daß dieser Wert allgemeingültig ist.

Möchten Sie beispielsweise mit dem 8253 den Kammerton a erzeugen, so müssen Sie den Wert 2712 in das Register mit der Portadresse 42hex schreiben, weil 1193180 geteilt durch 2712 den Frequenzwert 440 ergibt. Im Listing wird der Frequenzteiler für das 8253-Register berechnet, indem die interne Taktfrequenz von 1193180 Hz durch die Frequenz des gewünschten Tons geteilt wird. Dies läßt sich mit der Befehlsfolge

```
mov dx,12h
mov ax,34DCh
div di
```

realisieren. In diesem Beispiel wird die interne Taktfrequenz des 8253 als Hexadezimalwert 1234DChex in das Registerpaar DX:AX geschrieben und durch den Frequenzwert in DI geteilt. Das Ergebnis dieser Division können Sie dem Register AX entnehmen.

Um für die verschiedenen Töne eine konstante Länge zu garantieren, ist im Listing eine Zeitschleife notwendig. Eine Zeitverzögerung von 10 ms erreichen Sie bei einem PC mit einer Taktfrequenz von 4,772727 MHz über folgende Schleife:

```
milli: mov cx,2808
schleif: loop schleif
```

Der Wert 2808 errechnet sich aus der Taktfrequenz des PC und den Ausführungszeiten für die Assemblerbefehle. Der 8086/88-Prozessor benötigt für den Mov-Befehl vier Taktzyklen und für die Loop-Funktion entweder 17 Takte (wenn eine Verzweigung stattfindet) oder fünf Takte. Weil die Schleife immer verzweigt, bis CX den Wert 0 enthält, führt sie bei einem Schleifenwert von 2808

```
4+(2808-1)x17+5=47728
```

Taktzyklen aus. Bei einer Taktfrequenz von 4,772727 ist ein Taktzyklus

```
1 s/4772727=209,5 ns
```

lang. Damit ergibt sich eine Gesamtdauer der Schleife von

```
47728x209,5 ns = 10 ms
```

Wenn Ihr PC nicht mit einer Taktfrequenz von 4,772727 MHz arbeitet, müssen Sie dessen Frequenz in der Einheit Megahertz in folgende Formel einsetzen, um den korrekten Schleifenwert für eine Zeitdauer von 10 ms zu erhalten:

$$wert = (n / 10^{-4} - 9) / 17 + 1$$

Setzen Sie in diese Formel beispielsweise für n eine Taktfrequenz von 8 MHz ein, so müssen Sie im Listing in der Zeitschleife statt 2808 den Wert 4706 eintragen.

(Markus Zietlow/al)

```
[F77]    1: ;Assembler-Programm zur Demonstration der
[PL3]    2: ; Sound-Eigenschaften
[P3Ø]    3: ;
[PU3]    4: data segment para 'data'
[X76]    5: freq dw  98,11Ø,123,131,147,147,147,147,165
            ,131
[RG5]    6:      dw  196,165,147,165,131,196,165,147,14
            7,131
[M95]    7:      dw  131,131,131,123,123,123,123,11Ø,12
            3,11Ø
[N15]    8:      dw  98,123,147,147,147,131,131,131,131
            ,123
[UV3]    9:      dw  123,123,123,11Ø,123,11Ø, 98
[DMØ]   1Ø:      dw  Ø
[566]   11: laenge dw  12 dup (1ØØ), 1 dup (4ØØ), 4 dup
             (1ØØ)
[645]   12:      dw  1 dup (4ØØ),28 dup (1ØØ), 1 dup (4
            ØØ)
[DG1]   13: data ends
[P3Ø]   14: ;
[XT3]   15: code segment para 'code'
[MK2]   16: musik proc  far
[DU3]   17:      assume cs:code, ds:data
[ZA3]   18:      push  ax       ; Sichern
[DL2]   19:      push  bx       ;der
[G24]   2Ø:      push  cx       ;wichtigsten
[G94]   21:      push  dx       ;Speicherwerte
[Y55]   22:      mov   ax,seg data ; Datensegment
[NS4]   23:      mov   ds,ax    ;initialisieren
[ST4]   24:      lea   si,freq  ;Tabellenzeiger
[MT4]   25:      lea   bp,ds:laenge ;einrichten
[HR6]   26: spielen: mov di,[si] ; Folgenden Ton aus
[FD2]   27:                    ;Tabelle holen
[K55]   28:      cmp  di,Ø      ; Wenn die Melodie be-
[2V1]   29:                    ;endet ist,
[XW4]   3Ø:      je   schluss   ;Sprung zum Ende,
[B96]   31:      mov  bx,ds:[bp] ;ansonsten Tondauer
[ZV1]   32:                    ;ermitteln
```

```
[ZW5]   33:      mov  al,182    ; Ausgaberegister für
[EB5]   34:      out  43h,al    ;Timer-Baustein 8253
[GS1]   35:                     ;schreiben
[VU5]   36:      mov  dx,12h    ; Für den 2. Zeitgeber
[6P4]   37:      mov  ax,34dch  ;die benötigte
[X45]   38:      div  di        ;Frequenz bestimmen
[KK5]   39:      out  42h,al    ; Das niederwertige u.
[X85]   4Ø:      mov  al,ah     ;das höherwertige
[8N5]   41:      out  42h,al    ;Byte des errechneten
[FZ2]   42:                     ;Wertes an den 8253
[9C2]   43:                     ;übermitteln
[HC5]   44:      in   al,61h    ; Aktuellen Inhalt des
[CP5]   45:      mov  ah,al     ;Ausgaberegisters des
[895]   46:      or   al,3      ;PPI sichern und den
[BD4]   47:      out  61h,al    ;Lautsprecher
[X32]   48:                     ;einschalten
[466]   49: milli: mov cx,28Ø8  ; Ein Zeitintervall
[TV4]   5Ø: schleif: loop schleif ;warten
[AS5]   51:      dec  bx        ; Zähler dekrementieren,
[235]   52:      jnz  milli     ;bis er gleich Ø ist
[8C6]   53:      mov  al,ah     ; Ursprüngliche PPI-Werte
[TØ5]   54:      out  61h,al    ;wiederherstellen
[SU4]   55:      add  si,2      ; Zeiger der Ton- und
[SM3]   56:                     ;Lautstärken-Tabelle
[DS4]   57:      add  bp,2      ;um zwei erhöhen,
[XK5]   58:      jnz  spielen   ;bis Tabellenende er-
[K32]   59:                     ;reicht ist
[L65]   6Ø: schluss: pop dx     ; Die auf dem
[R25]   61:      pop  cx        ;Stapel gesicherten
[TZ2]   62:      pop  bx        ;Werte
[3V4]   63:      pop  ax        ;wiederherstellen
[1V4]   64:      mov  ah,4ch    ; Rückkehr in die
[WS3]   65:      int  21h       ;MS-DOS-Ebene
[SG5]   66: musik endp          ; Ende der Prozedur
[466]   67: code ends           ; Ende des Code-Segments
[1A5]   68:      end  musik     ; Ende des Programms
```

Listing. Dieses Beispiel zeigt, wie wenig Aufwand nötig ist, um auf dem PC »den Fuchs die Gans stehlen zu lassen«

Speicherverwaltung unter MS-DOS

Unter MS-DOS steht Ihnen ein Speicherplatz von maximal 640 KByte zur Verfügung, der direkt adressiert werden kann. Dieser Speicherbereich ist in ein festes Schema unterteilt, das MS-DOS auf verschiedene Arten verwaltet.

Der Speicher eines PC und AT wird von 00000 bis FFFFFhex adressiert. Beginnend von der Adresse 00000hex, enthält der erste Teil des Speichers die Tabellen für Interrupt-Vektoren, interne Variablenbereiche, Puffer und andere interne MS-DOS-Datenstrukturen. Es folgen die Gerätetreiber und der Programmcode des Betriebssystems und daran anschließend der freie Arbeitsspeicher. (Tabelle 1). Am oberen Ende des Speichers befindet sich das ROM-BIOS, darunter BIOS-Erweiterungen (zum Beispiel für Festplatten-Controller, EGA-Karte oder VGA-Karte). Unter diesem Bereich liegen die Speicherblöcke für die Bildschirmspeicher der einzelnen Grafikkarten.

Unterhalb dieser Speicherblöcke und oberhalb des Betriebssystems lagert die sogenannte »Transient Program Area« (TPA), hierbei handelt es sich um den für den Benutzer ansprechbaren Speicherbereich.

Die Größe der TPA ist von Rechnertyp zu Rechnertyp unterschiedlich. Sie hängt von der verwendeten MS-DOS-Version, der Anzahl der installierten Gerätetreiber, der RAM-Größe und anderen Konfigurationsparametern ab. Gerade aufgrund dieser Flexibilität in der Belegung benötigt MS-DOS eine dynamische Verwaltung des Speichers, um Speicherplatz variabel in unterschiedlich große Blöcke an einzelne Programme zuweisen zu können.

Der gesamte verfügbare Speicher von MS-DOS wird daher in unterschiedlich große Speicherblöcke aufgeteilt und an verschiedene Programme vergeben. Ein solcher Speicherblock muß mindestens 16 Byte lang sein und belegt maximal den gesamten zur Verfügung stehenden Speicher.

Die Speicherblöcke im PC besitzen unterschiedliche Größe

Sie können mit drei Funktionen des Betriebssystems auf die internen Routinen zur Speicherverwaltung zugreifen. Diese drei Funktionen sind über den Interrupt 21hex ansprechbar und erlauben die Belegung, die Freigabe und die Veränderung der Größe eines Speicherblocks. Tabelle 2 faßt diese Funktionen zusammen. Falls der Aufruf einer der Funktionen einen Fehler verursacht hat, ist das Carry-Flag gesetzt und der Fehlercode im Register AX zu finden. In Tabelle 3 sind die möglichen Fehlercodes zusammengefaßt.

Um die Übersicht über belegte und noch freie Speicherblöcke zu behalten, haben sich die Entwickler von MS-DOS den sogenannten »Memory Control Block« (MCB) einfallen lassen. Ein MCB ist eine Datenstruktur, die präzise Auskünfte über jeden verwalteten Speicherblock enthält.

Jeder MCB steht direkt vor dem von ihm kontrollierten Speicherbereich. Er enthält Informationen darüber, wieviel Platz der nachfolgende Speicherblock belegt und wo sich der nächste MCB befindet. Ein MCB belegt 16 Byte, seine Bedeutung ist in Tabelle 4 dargestellt.

Wichtig ist zunächst einmal die Identifikation eines MCB. Hier hat sich einer der Entwickler von MS-DOS, der die Initialen »M.Z.« trägt, verewigt. Wenn dem MCB mindestens ein weiterer MCB folgt, enthält das Identifikationsbyte des MCB den Buchstaben »M«, ansonsten den Buchstaben »Z«. Das nächste Feld, die Adresse des »PSP« (»Program Segment Prefix«), ist nur von Interesse, wenn mit dem MCB ein »Environment« verwaltet wird. Wichtiger ist das nächste Feld, das die Größe des hinter dem MCB belegten Speichers in 16-Byte-Abschnitten wiedergibt. Hieraus ermitteln Sie nämlich einen Zeiger auf den nächsten MCB, sofern der aktuelle MCB nicht ein »Z« als Identifikation trägt, denn dann wäre er ja der letzte MCB in der Kette.

Anhand der MCBs kann MS-DOS sehr präzise ermitteln, welche Speicherblöcke belegt sind und wo sich noch freier Speicherplatz befindet. Daher ist es besonders aufschlußreich, sich einmal die Belegung des Speichers durch die MCBs anzeigen zu lassen.

Sie müssen dazu allerdings wissen, wie man die Adresse des ersten MCB ermittelt. Dabei bedienen Sie sich einer nicht dokumentierten MS-DOS-

Adresse	Funktion
00000-0007F	BIOS-Unterbrechungsaufrufe
00080-003FF	MS-DOS-Unterbrechungsaufrufe
00400-004FF	ROM-BIOS-Datensegment
00500-005FF	für MS-DOS und Basic reserviert
00600-nnnnn	MS-DOS-Betriebssystem
nnnnn-9FFFF	Freier Arbeitsspeicher
B0000-B0FFF	Monochromadapter
B0000-BFFFF	Hercules-Karte
B8000-BBFFF	CGA-Karte
A0000-BFFFF	EGA-Karte
A0000-AFFFF	VGA-Karte
C0000-FFFFF	ROM-Bereich für Basica, BIOS und BIOS-Erweiterungen

Tabelle 1. Die wichtigsten Speicheradressen des PC und AT

48hex:	Speicherblock belegen
Eingabe:	AH = 48hex
	BX = Anzahl der zu belegenden 16-Byte-Abschnitte
Ausgabe:	BX = Anzahl der belegten 16-Byte-Abschnitte
	Carry-Flag zeigt Fehler an, dann: AX = Fehlercode
49hex:	Speicherblock freigeben
Eingabe:	AH = 49hex
	ES = Segmentadresse des freizugebenden Speicherblocks
Ausgabe:	Carry-Flag zeigt Fehler an, dann: AX = Fehlercode
4Ahex:	Größe des Speicherblocks verändern
Eingabe:	AH = 4Ahex
	ES = Segmentadresse des freizugebenden Speicherblocks
	BX = Neue Anzahl der zu belegenden 16-Byte-Abschnitte
Ausgabe:	BX = Anzahl der belegten 16-Byte-Abschnitte
	Carry-Flag zeigt Fehler an, dann: AX = Fehlercode

Tabelle 2. Die MS-DOS-Funktionen zur Speicherverwaltung

Funktion, nämlich der Funktion 52hex des Interrupts 21hex. Diese Funktion liefert als Resultat die Adresse des sogenannten »DIB« (DOS Information Block) im Registerpaar ES:BX zurück. Neben einigen anderen hochinteressanten Informationen, in die der DIB Einblick gewährt, befindet sich dort an der Offsetadresse - 4 auch ein Zeiger auf den ersten MCB. Wenn Sie dieses Feld auslesen, ist es einfach, die gesamte Kette der MCBs und der dazugehörigen Speicherblöcke zu untersuchen.

Das erledigt das Programm »showmcbs.pas« (Listing 1), das unter Turbo Pascal 4.0 oder 5.0 lauffähig ist. Die Änderungen für Turbo Pascal vor der Version 4.0 bleiben minimal und sind im Quelltext dokumentiert. Das Programm zeigt die Belegung des Speichers anhand der MCBs und der dazugehörigen Speicherblöcke.

MS-DOS kennt drei Arten, den Speicherplatz zu verwalten

MS-DOS bearbeitet mit Hilfe der MCBs mögliche Speicherplatzanforderungen. Wird zum Beispiel von einem Programm zusätzlicher Speicherplatz angefordert, so durchsucht das Betriebssystem die Kette der MCBs, bis ein entsprechender Speicherblock gefunden ist. Bei dieser Suche beherrscht das Betriebssystem drei unterschiedliche Belegungsstrategien.

Bei der ersten Strategie, der Belegungsstrategie 0, beginnt MS-DOS die Suche bei dem am weitesten unten liegenden Speicherblock, also bei möglichst kleinen Speicheradressen. Der erste Block, der mindestens genauso groß ist wie der angeforderte Speicherblock, wird dann belegt. Diese Methode wird »first fit« (»erstes Zutreffen«) genannt.

Bei der zweiten Strategie, die mit der Nummer 1 bezeichnet wird, durchsucht MS-DOS alle Speicherblöcke nach dem Block, der die Anforderung am geringsten übersteigt, aber dennoch groß genug ist. Diese Methode wird aufgrund ihrer Wirtschaftlichkeit auch »best fit« (»bestes Zutreffen«) genannt.

Die letzte Strategie schließlich, die Methode 2, beginnt die Suche bei dem am weitesten oben liegenden Speicherblock, also bei möglichst großen Speicheradressen. Der erste Block, der mindestens genauso groß ist wie der angeforderte Block, wird dann belegt. Diese Methode heißt »last fit« (»letztes Zutreffen«). Diese Methode ist die Umkehrung von Belegungsstrategie 0.

MS-DOS verwendet standardmäßig die Belegungsstrategie 0, die eine schnellstmögliche Vergabe von Speicherblöcken gewährleistet. Vor allem für Besitzer von PCs mit wenig Speicherplatz kann es aber sinnvoll sein, diese Belegungsstrategie zu ändern. Hierzu ein Beispiel: Angenommen, es sind bei Ihrem PC noch 64 KByte Speicher frei, die beispielsweise auf zwei freie Speicherblöcke verteilt sind. Der erste Block soll dabei 48 KByte enthalten, der zweite die verbleibenden 16 KByte. Wenn nun ein Programm einen Speicherblock von beispielsweise 14 KByte anfordert, weist MS-DOS ihm nach der Belegungsstrategie 0 den ersten Speicherblock zu, der ausreichend groß ist. Daher wird der 48-KByte-Block in zwei Blöcke unterteilt: einen belegten 14-KByte-Block und einen freien 34-KByte-Block. Wenn nun ein anderes Programm beispielsweise einen neuen Speicherblock von 45 KByte anfordert, kann MS-DOS diesen Auftrag nicht ausführen, da nur noch ein 34-KByte- und ein 16-KByte-Block zur Verfügung stehen. Hätte MS-DOS jedoch Strategie 1 verwendet, wäre das Problem nicht aufgetreten: Die 14-KByte-Anforderung hätte den 16-KByte-Block belegt und der große 48-KByte-Speicherblock wäre für die 45-KByte-Anforderung noch unbelegt und unzerteilt gewesen.

Daher ist ein Programm hilfreich, mit dessen Hilfe die Parameter des Betriebssystems so geändert werden, daß immer Belegungsstrategie 1 verwendet wird. Genau das geschieht mit dem Programm »setram.asm«, das in Listing 2 abgedruckt ist. Es aktiviert die Belegungsstrategie 1 und gibt eine entsprechende Meldung aus.

(Martin Althaus/wn)

Code	Bedeutung
7	MCBs sind zerstört
8	Speicherplatz reicht nicht aus
9	ungültige Segmentadresse

Tabelle 3. Die möglichen Fehlercodes

Byte	Inhalt
00h	Identifizierungscode. Hat den Wert 77 ("M"), wenn weitere MCBs folgen, ansonsten den Wert 90 ("Z")
01h	niederwertiges Byte der Adresse des PSP des zugehörigen Programms
02h	höherwertiges Byte der Adresse des PSP des zugehörigen Programms
03h	niederwertiges Byte der Größe des Speicherblocks in 16-Byte-Abschnitten
04h	höherwertiges Byte der Größe des Speicherblocks in 16-Byte-Abschnitten
05h-10h	reserviert

Tabelle 4. Der Aufbau eines MCB

```
[TØ3]    1: program showmcbs;
[ØØØ]    2:
[FN5]    3: (* Stellt die Kette der MCBs auf dem
[KB2]    4:    Bildschirm dar
[2W4]    5:    entwickelt von Martin Althaus
[7Y5]    6:    (c) Redaktionsbüro Everts & Hagedorn
[F6Ø]    7: *)
[ØØØ]    8:
[S62]    9: uses dos,crt;
[TØ7]   1Ø: (* muß bei Benutzung von Turbo Pascal vor
[1V2]   11:    Version 4.Ø durch
[ØØØ]   12:
[STØ]   13:    type
[KS2]   14:      register = record
[MH3]   15:        case boolean of true:
[N17]   16:          (ax,bx,cx,dx,bp,si,di,ds,es,flags:in
           teger);
```

```
[WUØ]   17:          false
[HU4]   18:          (al,ah,bl,bh.cl,ch,dl,dh:byte);
[ZNØ]   19:        end;
[ØØØ]   2Ø:
[GX2]   21:  ersetzt werden *)
[STØ]   22: type
[4U2]   23:  pointer = record
[DK3]   24:              segment,offset:word;
[ZNØ]   25:            end;
[GL3]   26:  vierstr = string[4];
[444]   27:  einsstring = string[1];
[RNØ]   28: var
[YT2]   29:  cpu: registers;
[NG3]   3Ø:  taste,mcb_code: char;
[KA4]   31:  dib_adresse,mcb: pointer;
[FB4]   32:  mcb_nummer,groesse: word;
[ØØØ]   33:
```

```
[E6Ø]   34: (*
[187]   35:  Funktion zur Umwandlung einer Zahl in eine
[3Ø5]   36:  vierstelige Hexadezimalzahl
[F6Ø]   37: *)
[ØØØ]   38:
[3U6]   39: function make_hex(wert: word):vierstr;
[6XØ]   4Ø:  const
[ØX5]   41:   hex_ziffern: array[Ø..15] of char =
[3P7]   42:     ('Ø','1','2','3','4','5','6','7','8','9'
        ,'A','B','C','D','E','F');
[RNØ]   43:  var
[9E2]   44:   help: vierstr;
[EZ1]   45:   index: byte;
[C82]   46:   teiler: word;
[MUØ]   47:  begin
[3T1]   48:   help:='ØØØØ';
[9Z3]   49:   for index:=3 downto Ø do
[MUØ]   5Ø:    begin
[A75]   51:     teiler:=wert shr (index shl 2);
[SØ6]   52:     wert:=wert-(teiler shl (index shl 2));
[YC6]   53:     help[4-index]:=hex_ziffern[teiler];
[ZNØ]   54:    end;
[MP2]   55:   make_hex:=help;
[ZNØ]   56:  end;
[ØØØ]   57:
[E6Ø]   58: (*
[KØ7]   59:  Funktion zum Einrichten eines variablen
[ZU1]   6Ø:  Tabulators
[F6Ø]   61: *)
[ØØØ]   62:
[5Z6]   63: function tab(position:byte):einsstring;
[MUØ]   64:  begin
[R84]   65:   gotoxy(position,wherey);
[L31]   66:   tab:='';
[ZNØ]   67:  end;
[ØØØ]   68:
[E6Ø]   69: (*
[D76]   7Ø:  Hauptprogramm: verfolgt die MCBs und
[TF6]   71:  stellt den Inhalt auf dem Bildschirm dar.
[F6Ø]   72: *)
[ØØØ]   73:
[MUØ]   74: begin
[DB1]   75:  clrscr;
[XF1]   76:  writeln;
[MU7]   77:  writeln(tab(17),'-------------------------
        ----------------');
[5T7]   78:  writeln(tab(17),'----            SHOWMCBS V
        1.Ø            ----');
[G1A]   79:  writeln(tab(17),'--- (c) Redaktionsbüro Ev
        erts & Hagedorn ---');
[MU7]   8Ø:  writeln(tab(17),'-------------------------
        ----------------');
[XF1]   81:  writeln;
[1V1]   82:  cpu.ah:=$52;
[XV1]   83:  msdos(cpu);
[H75]   84:  dib_adresse.segment:=cpu.es-1;
[245]   85:  dib_adresse.offset:=cpu.bx+12;
[PX9]   86:  mcb.offset :=mem[dib_adresse.segment:dib_a
        dresse.offset]+
[EM8]   87:                256*mem[dib_adresse.segment:d
        ib_adresse.offset+1];
[PGA]   88:  mcb.segment:=mem[dib_adresse.segment:dib_a
        dresse.offset+2]+
[GM8]   89:                256*mem[dib_adresse.segment:d
        ib_adresse.offset+3];
[YC2]   9Ø:  mcb_nummer:=1;
[ED8]   91:  writeln('┌──────┬──────┬──────┬──────┬──────
        ──────┐');
[TAF]   92:  writeln('│ Nummer │ Code │ Adresse de
        s    │ Größe des      │ Größe des
        │');
[TVE]   93:  writeln('│ des    │ des  │ Speicherbl
        ocks │ Speicherblocks │ Speicherblocks
        │');
[RGC]   94:  writeln('│ MCBs   │ MCBs │
        │ (Paragraphen)  │ (Bytes)
        │');
[9E8]   95:  writeln('├──────┼──────┼──────┼──────┼──────
        ──────┤');
[P31]   96:  repeat
[1T3]   97:   write('│  ',mcb_nummer);
[WB7]   98:   mcb_code:=char(mem[mcb.segment:mcb.offset
        ]);
[AE4]   99:   write(tab(12),'│  ',mcb_code);
[F57]   1ØØ:  write(tab(21),'│  ',make_hex(mcb.segment+
        1),':',
[ØH4]   1Ø1:                        make_hex(mcb.offset),
        'hex');
[KE7]   1Ø2:  groesse:=(mem[mcb.segment:mcb.offset+3]+2
        56*
[8B5]   1Ø3:             mem[mcb.segment:mcb.offset+4]);
[PC4]   1Ø4:  write(tab(4Ø),'│  ',groesse);
[5W6]   1Ø5:  write(tab(59),'│  ',longint(groesse) shl
        4);
[ZG3]   1Ø6:  writeln(tab(77),'│');
[XF1]   1Ø7:  writeln;
[ØM2]   1Ø8:  write('<Taste>');
[KN2]   1Ø9:  taste:=readkey;
[YY2]   11Ø:  gotoxy(1,wherey);
[DH2]   111:  writeln('        ');
[C24]   112:  gotoxy(wherex,wherey-2);
[M36]   113:  mcb.segment:=mcb.segment+groesse+1;
[4A4]   114:  mcb_nummer:=mcb_nummer+1;
[993]   115: until mcb_code<>'M';
[6D8]   116: writeln('└──────┴──────┴──────┴──────┴──────
        ──────┘');
[KNØ]   117: end.
```

Listing 1. Das Turbo-Pascal-Programm »showmcbs.pas« zeigt eine typische Speicherbelegung

```
[P3Ø]    1: ;
[EØ2]    2: ; SETRAM.ASM
[4X5]    3: ; entwicklt von Martin Althaus 1989
[5E6]    4: ; (c) Redaktionsbüro Everts & Hagedorn
[5U4]    5: ;---------------------------------------
[P3Ø]    6: ;
[6N6]    7: ; In diesem Segment wird nach Aufruf des
[X56]    8: ; Programms der Stapel untergebracht
[P3Ø]    9: ;
[368]    1Ø: ;Segment soll an einem Paragraphen beginnen
         stapel
[Z63]    11:     segment para stack
[HØ7]    12: ;512 Bytes reservieren,reicht für 256 Words
[682]    13:          dw 1ØØh dup (?)
[8R2]    14: ;Segment beenden
[742]    15: stapel    ends
[P3Ø]    16: ;
[5U4]    17: ;---------------------------------------
[P3Ø]    18: ;
[EW5]    19: ; In diesem Segment werden die Daten
[5S2]    2Ø: ; untergebracht
[P3Ø]    21: ;
[877]    22: ;Segment soll an einem Paragraphen beginnen
[5A4]    23: daten     segment para 'data'
[EL2]    24: msg1      db Øah,Ødh
[T25]    25:           db 'DOS arbeitet zur Zeit mit '
[2C2]    26:           db 'Strategie '
[NW1]    27: strat     db Ø
[P95]    28:           db ' zur Belegung des Speichers.'
[SC1]    29:           db Øah,Ødh
[K84]    3Ø:           db 'Die Strategie für die '
[YF4]    31:           db 'Belegung des Speichers '
[M35]    32:           db 'wurde jetzt auf 1 gesetzt.'
[6B2]    33:           db Øah,Ødh,Øah,'$'
[SR2]    34: ; Segment beenden
[ZZ1]    35: daten     ends
[P3Ø]    36: ;
[5U4]    37: ;---------------------------------------
[P3Ø]    38: ;
[SY6]    39: ; In diesem Segment wird der Programmcode
[LT2]    4Ø: ; untergebracht.
[P3Ø]    41: ;
[877]    42: ;Segment soll an einem Paragraphen beginnen
[PN4]    43: programm  segment para 'code'
[WK6]    44: ;Zeige dem Assembler an, welches Segment
[XY6]    45: ;sich auf welches Segmentregister bezieht
[WR6]    46:    assume cs:programm, ds:daten, ss:stapel
[DC5]    47: ;hier beginnt das Hauptprogramm
[RL2]    48: start     proc far
[Y37]    49: ;lade Adr. des Datensegments und übertrage
[B72]    5Ø:  mov   ax,daten
[E44]    51:  mov   ds,ax       ;sie nach DS
[V95]    52:  mov   ah,58h      ;DOS-Funktion 58hex
[134]    53:  xor   al,al       ;lösche AL
[B35]    54:  int   21h         ;lese Strategie aus
[BH4]    55:  add   al,48       ;justiere ASCII
[TK6]    56:  mov   strat,al    ;speichere Strategie ab
[RU5]    57:  mov   dx,offset msg1  ;lade Adresse
[246]    58:  mov   ah,9        ;gebe sie über Funktion 9
[1Z5]    59:  int   21h         ;von Interrupt 21hex aus
[3W5]    6Ø:  mov   ah,58h      ;lade Funktionsnummer
[R35]    61:  mov   al,1        ;Strategie setzen
[CC5]    62:  mov   bx,1        ;Strategie: best fit
[9F5]    63:  int   21h         ;setze die Strategie
[DT6]    64:  mov   ax,4cØØh    ;beende das Programm über
[YØ4]    65:                    ;Funktion 4Ch ohne Fehler
[LY1]    66:                    ;code über
[253]    67:  int   21h         ;das DOS
[TU6]    68: start   endp       ;hier endet Hauptprogramm
[ØØØ]    69:
[TF6]    7Ø: programm  ends     ;hier endet das Segment
[N56]    71:  end  start        ;hier endet das Listing,
[D43]    72:                    ;es soll bei START
[7W2]    73:                    ;gestartet werden
```

Listing 2. »setram.asm« erzeugt eine bessere Speicherbelegungsstrategie als standardmäßig vorhanden

Optimale TSR-Programmierung in Assembler

Das Schreiben speicherresidenter Programme ist vom Prinzip her einfach, da diese sogenannten TSR-Programme alle den gleichen Aufbau besitzen. Die Beachtung einiger Nebenbedingungen führt dann auch schnell zum Erfolg. Was es im einzelnen zu berücksichtigen gilt, erfahren Sie in dieser Werkstatt.

Wollen Sie TSR-Dienstprogramme schreiben, die nicht allzuviel Platz des kostbaren Arbeitsspeichers belegen, so ist dies eigentlich nur mit Hilfe der Assemblersprache realisierbar. Gleichzeitig läßt sich der Code bezüglich des Zeitverhaltens optimieren und erlaubt Zugriffe zum Computer auf unterster Ebene. Um wirklich brauchbare Programme zu erhalten, sind aber einige Punkte zu beachten, die als »Nebenbedingungen« durch das Betriebssystems und das BIOS diktiert werden.

Für das Verständnis der TSR-Programmierung sind Grundkenntnisse in der Verwaltung eines laufenden Programms unter dem jeweiligen Betriebssystem von Vorteil. Mit Hilfe dieser Kenntnisse muß ein Programmumfeld geschaffen werden, unter dem das Programm auch dann lauffähig bleibt, wenn es als Interrupt-Handler oder als aufrufbares Dienstprogramm (zum Beispiel über eine Hotkey-Taste) aktiviert wird.

Das größte Problem in diesem Zusammenhang besteht darin, daß das Betriebssystem MS-DOS ein reines »Single-Task-Betriebssystem« ist. »Single-Task« bedeutet, daß das Betriebssystem immer nur in der Lage ist, genau ein Programm (Fachbegriff: Task) auszuführen. Daher sind bei MS-DOS auch keinerlei Vorkehrungen getroffen, um konkurrierende Prozesse zu bedienen. Konkurrierend heißt hier, daß das eine Programm beispielsweise gerade eine Bildschirmausgabe vorsieht und dann vom zweiten Programm unterbrochen wird, das ebenfalls eine Ausgabe auf den Bildschirm vornehmen will. Unter MS-DOS führt dies zu nicht vorhersehbaren Ergebnissen bis hin zum Systemabsturz.

Sind Systemfunktionen in der Lage, solche Mehrfachzugriffe zu verkraften, so nennt der Fachmann sie »reentrant«, eine Eigenschaft, die MS-DOS gänzlich fehlt.

Die einzigen getroffenen Vorkehrungen beschränken sich darauf, daß MS-DOS ein Statusbyte verwaltet, das anzeigt, ob und wie viele verschachtelte MS-DOS-Aufrufe aktiv sind. Die Speicherstelle, an der dieses Byte abgelegt ist, erhalten Sie durch Aufruf der Funktion 34hex des MS-DOS-Interrupt 21hex. Die Adresse wird in dem Registerpaar ES:BX zurückgeliefert. Dieses Flag alleine reicht aber nicht aus, um Kollisionen des eigenen Programms mit MS-DOS zu vermeiden.

MS-DOS setzt Zeichen, wenn es mit sich selbst beschäftigt ist

Befinden Sie sich noch in der Befehlseingabe für MS-DOS »command.com«, so ist dieses Statusbyte immer ungleich Null und ihre eigene Routine würde nie aufgerufen. Der Kommandoprozessor löst aber ständig den Interrupt 28hex aus, wenn er nichts zu tun hat. Dieser Interrupt ist ein sicherer Zeitpunkt, um speicherresidente Programme mit MS-DOS-Aufrufen abzuarbeiten.

Ähnlich wie das Betriebssystem verhalten sich die BIOS-Routinen, die auch nicht reentrant sind. Um hier Abhilfe zu schaffen, müssen Sie sich für die von Ihnen benötigten Routinen selbst ein solches Statusbyte deklarieren.

Folgendes Programm wird in den gewünschten Interrupt eingeklinkt und kümmert sich um das Statusbyte. Mit diesem stellen Sie dann fest, ob diese BIOS-Funktion bereits ausgenutzt wird:

```
inc cs:biosaktiv
call AltenBiosInterrupt
dec cs:biosaktiv
iret
```

Dieser Trick wird zum Beispiel auch von diversen Debuggern genutzt. Die Verwaltung der passenden Programmumgebung (Prozessorregister, »Program Segment Prefix« (PSP) und »Disk Transfer Adress« (DTA)) muß also von Ihnen vorgenommen werden. Nur unter dieser Voraussetzung sind lauffähige TSR-Programme zu schreiben, die problemlos funktionieren und das Hauptprogramm nicht stören.

Eine Weisheit, die sich inzwischen herumgesprochen hat, ist, daß zumindest alle Prozessorregister gesichert werden müssen, um beim Verlassen des TSR-Programms die Register wieder mit den alten Werten zu versehen. So arbeitet das unterbrochene Programm ohne Veränderung an der Stelle weiter, an der es unterbrochen wurde. Zweck dieser Handlungsweise ist es, die unmittelbare Programmumgebung des unterbrochenen Programms nicht zu verändern. Sie müssen also dafür Sorge tragen, daß das unterbrochene Pro-

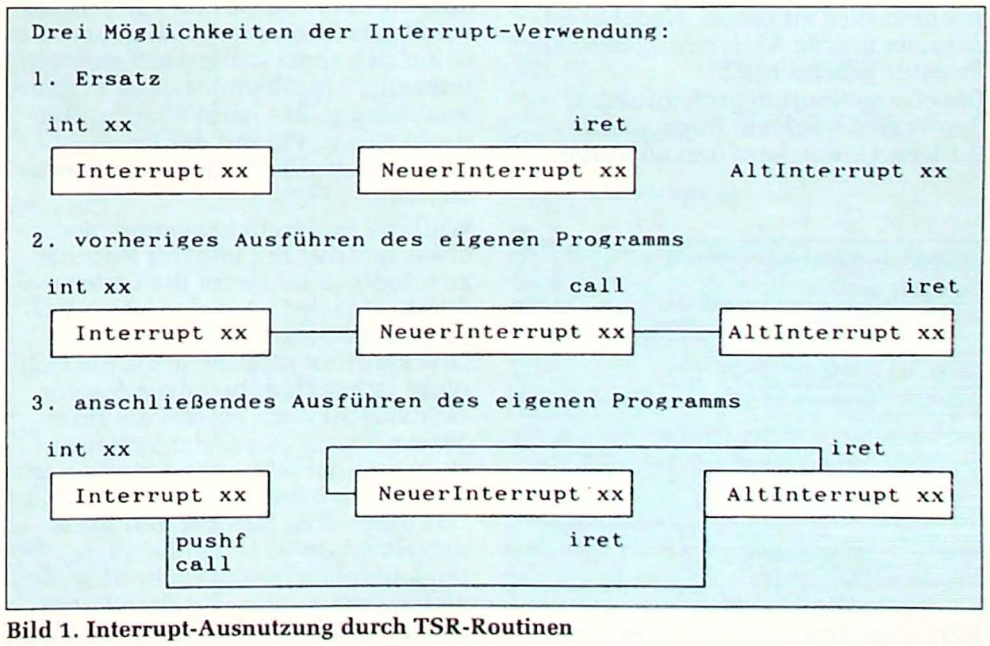


Bild 1. Interrupt-Ausnutzung durch TSR-Routinen

gramm von dieser Unterbrechung gar nichts merkt. Es spielt hierbei keine Rolle, ob das TSR-Programm durch Tastendruck oder durch einen Interrupt aktiviert wird. Bei der Aktivierung durch Tastendruck ist entweder der Interrupt 9 abzufangen (Prozessorebene) und durch eine eigene Routine zu ersetzen oder aber der Interrupt 16hex (BIOS-Interrupt).

Bevor Sie einen Interrupt belegen, müssen Sie den Verbleib seiner bisherigen Funktionen klären

Da bei der TSR-Programmierung eigentlich immer mindestens ein Interrupt beteiligt ist, der »verbogen« wird, ist sehr genau zu überlegen, was Sie mit dem alten Interrupt anfangen. Um dies zu beurteilen, ist es wichtig abzuklären, ob das Betriebssystem den Interrupt unbedingt benötigt (zum Beispiel den Interrupt 8) und der Original-Interrupt daher auf jeden Fall aufgerufen werden muß, ob Sie den Interrupt durch eine eigene Routine vollständig ersetzen (zum Beispiel den Print-Screen-Interrupt 5) oder ob Sie Ihre Routine mit »einhängen«.
Denkbar ist zum Beispiel ein vollständiger Ersatz des Interrupt 5 (Hardcopy des Bildschirms) durch eine eigene Routine, die dann die alte Routine nicht mehr aufruft, da sonst der Bildschirm zweimal gedruckt würde. Soll die neue Routine nur zusätzlich aktiviert werden, so ist zu überlegen, ob der alte Interrupt vorher oder nachher aufgerufen werden soll. Vorher ist zum Beispiel beim Timer-Interrupt (Interrupt 8 oder 1Chex) denkbar, um Zeitproblemen aus dem Weg zu gehen. Nachher ist dagegen für die Aktivierung über die Tastatur gebräuchlich.
Die eigene Routine prüft zunächst den Tastendruck auf ihren »Hotkey« (Listing 1), um dann den normalen Tastatur-Interrupt zu beenden, falls die Taste nicht die gewünschte ist oder aber die zu aktivierende Routine aufruft. Die beiden letzten Verfahren werden mit »Chaining« (deutsch: verketten) bezeichnet (Bild 1), da eine Kette von einer Interrupt-Behandlungsroutine (Fachbegriff: Interrupt-Handler) zur nächsten gebildet wird, die alle über den gleichen Interrupt aktiviert werden.

```
Hilfreiche MS-DOS-Funktionen:

Funktion  Aufgabe
1Ah       Setzen der DTA-Adresse
          Eingabe: AH    = 1Ahex
                   DS:DX = Adresse DTA

2Fh       Lesen der DTA-Adresse
          Eingabe: AH    = 2Fhex
          Ausgabe: ES:BX = Adresse DTA

34h       Ermitteln des INDOSFLAG
          Eingabe: AH    = 34hex
          Ausgabe: ES:BX = Adresse INDOS-Flag

50h       Setzen des PSP
          Eingabe: AH    = 50h
                   BX    = Segmentadresse PSP

51h       Lesen des PSP
          Eingabe: AH    = 51h
          Ausgabe: BX    = Segmentadresse PSP
```

Bild 2. Hilfreiche MS-DOS-Funktionen

Ein weiteres Problem von TSR-Routinen ist die Stack-Benutzung. Die Benutzung des Stacks des unterbrochenen Programms ist eine Tatsache, die sehr schnell zum Dilemma führt, wenn man bedenkt, daß zumindest alle Prozessorregister gesichert werden müssen. Die Anzahl der freien Bytes auf dem Stack ist generell nicht vorhersehbar, so daß das unterbrochene Programm eventuell nur noch sehr wenige davon zur Verfügung hat. Die Folge ist in der Regel ein merkwürdiges Benehmen des Computers, da Speicherplätze überschrieben werden, die anderweitig belegt sind. Um nun den fremden Stack nicht unnötig zu belasten, verwenden Sie lieber einen eigenen Stackbereich.

Ohne eigenen Stackbereich könnte ihr Programm im Computer ein Chaos hinterlassen

Gleichzeitig deklarieren Sie die Variable »aktiv«. Anhand dieser Variablen erkennt Ihr TSR-Programm, ob es bereits aktiv ist. Die Notwendigkeit eines solchen Flags ist an einem sehr einfachen Beispiel zu verdeutlichen: Stellen Sie sich vor, Sie haben eine eigene Prozedur für Hardcopies vom Bildschirm geschrieben und diese auf den sonst völlig problemlosen Interrupt 5 (problemlos, weil systemunabhängig, das heißt nicht zeitkritisch) gelegt, um mit der Print-Screen-Taste die Aktivierung vorzunehmen.
Weil der Rechner aber gerade noch etwas anderes mit höherer Priorität zu erledigen hat, wird Ihr Tastendruck zunächst einmal in den Tastaturpuffer übernommen, und der Drucker rührt sich nicht. Da Sie sich nicht sicher sind, betätigen Sie vorsichtshalber noch einmal die Print-Screen-Taste, womit der Systemabsturz erreicht wäre. Die Situation ist nun die, daß Sie Ihre Hardcopy-Routine durch Ihre Hardcopy-Routine unterbrechen. Weil der benutzte Speicherplatz für die Daten aber der gleiche ist, werden die alten Daten zerstört (insbesondere auch die Rücksprungadresse und die Prozessorregister), und das System kommt gründlich durcheinander (dies ist auch ein Beispiel für die Vorgänge bei »nicht reentrant« programmierten Funktionen).
Nach dem Sichern der Register werden die Rücksprungadresse und die Flags des unterbrochenen Programms vom Stack entfernt und gesichert. Sollte es erforderlich sein (es gibt einige ganz hartnäckige Fälle), so sind zusätzlich der Programmvorspann (PSP) und die Plattenpuffer (DTA) auf TSR-eigene Datenbereiche einzustellen. Um die hierfür nötigen Informationen zu erhalten, verwenden Sie die Funktionen 51 oder 62hex, um das PSP zu ermitteln, und die Funktion 50hex, um den PSP zu setzen. Die DTA erhalten Sie durch die Funktion 2Fhex und setzen sie über die Funktion 1Ahex (alles Funktionen des MS-DOS-Interrupt 21hex).
Nach der Programmausführung werden die Register zurückgesetzt, der Stack wieder zurückgestellt und die Rücksprungadressen auf den alten Stack geschoben. Mit »iret« geht es dann zurück ins unterbrochene Programm.
Die in den letzten Zeilen beschriebenen Vorgänge sind doch ziemlich komplex. Lesen Sie sie ruhig noch ein zweites Mal, und betrachten Sie parallel dazu Listing 2, eine erste Version des Hardcopy-Programms, das den Bildschirminhalt in eine Datei schreibt.

Für Programme mit Bildschirmausgaben ist die Wiederherstellung des alten Zustandes ein Muß

Möchten Sie Meldungen auf den Bildschirm ausgeben, so ist der alte Bildschirminhalt zu sichern und nach Beendigung des Programms wiederherzustellen. Hierzu gehört selbstverständlich auch die Stellung des Cursors (aufpassen, wenn Sie mit BIOS-Funktionen arbeiten).
Das Restaurieren des Bildschirms ist ein wichtiger Bestandteil der guten TSR-Geister und der Grund dafür, daß die meisten TSR-Programme nur im Textmodus funktionieren. Bei EGA-Auflösung müßten Sie sonst bis zu 256 KByte Speicher nur zum Sichern des Bildschirminhaltes bereitstellen. Diese Menge an freiem Speicherplatz steht heutzutage jedoch nur noch in den seltensten Fällen zur Verfügung.
Wie bereits erwähnt, ist der hier benutzte Interrupt 5 ein besonders unkritischer Interrupt, der sich hervor-

ragend zum Testen von TSR-Programmen eignet. Er wird nicht vom System benutzt, ist nicht zeitkritisch und für das Betriebssystem nicht lebensnotwendig. Hinzu kommt, daß Sie diesen Interrupt über Tastendruck auslösen. Das sonst etwas schwierige Debuggen von TSR-Programmen erweist sich hierbei ebenfalls als problemlos.
Laden Sie das TSR-Programm, und starten Sie den Debugger. Assemblieren Sie den Befehl »int 5«, und durchlaufen Sie schrittweise Ihr Programm (mit dem Befehl »t«). Sollte es vorkommen, daß Ihr Programm unter der Kontrolle des Debuggers einwandfrei läuft, normalerweise aber nicht, so ist sicherlich ein wiederholter BIOS- oder MS-DOS-Aufruf schuld, den Sie nicht abgefangen haben.

Durch Assemblieren mit dem MASM oder TASM wird der Quellcode lauffähig

Um die Listings lauffähig zu machen, müssen Sie folgende Schritte durchführen: Zunächst muß die Assemblerprozedur in eine Objektdatei übersetzt werden. Dies ist entweder mit dem Makroassembler MASM oder mit dem Turboassembler TASM durchzuführen. Die so entstandenen Objektdateien werden noch gebunden. Ein Beispiel:

```
MASM test.asm
LINK test.asm
exe2bin test.asm
```

oder

```
TASM test.asm
TLINK test /t
```

Anschließend installieren Sie das Programm durch einfachen Aufruf aus der MS-DOS-Ebene im Speicher.
Besonders kritisch wird die TSR-Programmierung, wenn zeitkritische Interrupts (zum Beispiel der Interrupt 1Chex) verwendet werden, da diese Unterbrechungen zyklisch aufgerufen werden (in diesem Fall ungefähr alle 18 Millisekunden), wobei nur eine geringe Zeitspanne zwischen den Aufrufen für eigene Zwecke zur Verfügung steht.
Ist Ihr Programm zu langsam, wird es schon wieder aufgerufen, bevor es beendet wurde. Dieses Problem ist zu umgehen, indem das Interrupt-Programm während der Ausführung die Original-Interrupts wieder herstellt und erst bei Beendigung den Interrupt wieder auf sich selbst setzt. So schlagen Sie gleich zwei Fliegen mit einer Klappe: Zeitbeschränkungen werden aufgehoben, und das Programm findet die originale Interrupt-Umgebung (fast) ohne Einschränkungen.
Die Bestimmung des Bildschirmtyps bei der Installation (Listing 3) ist für eine allgemeinere Verwendung des Programms bestimmt, so daß Sie wieder ein bißchen rechnerunabhängiger werden.
Mit der Funktion und dem Umfeld der TSR-Programme haben Sie sich nun beschäftigt, aber was ist bei der Installation eines solchen Programms alles zu beachten? Mit Hilfe des Listings 2 (Hardcopy in Datei) werden diese Schritte deutlich: Das Bestimmen der Grafikkarte ist nicht unbedingt notwendig, aber ganz reizvoll. Anschließend wird der zu ersetzende Interrupt gespeichert und die neue Interrupt-Prozedur gesetzt. Durch den Aufruf von »Keep« (Interrupt 27hex) wird das Programm beendet und verbleibt im Speicher.

Am Beispiel eines Hardcopy-Programms wird die Methodik eines TSR-Programms schnell klar

Die bisherigen Programme dienten überwiegend der Vermittlung von Grundlagen und dem Gewinnen von Erfahrung mit TSR-Routinen. Nun folgt mit Listing 4 noch ein Anwendungsbeispiel, das stilistisch zwar etwas mager ausgefallen ist, aber durchaus eine Daseinsberechtigung hat.

Listing 4 hat, kurz zusammengefaßt, folgende Funktion: Während der Installation wird der Bildschirm geprüft, der Original-Interrupt 28hex gespeichert, der neue Interrupt 5 gesetzt (vollständiger Ersatz) und der Interrupt 28hex auf die neue Hardcopy-Routine »umgebogen« (Einhängen). Durch die Verwendung des Interrupt 28hex umgehen Sie die Probleme der Version 1 (Listing 2). Beim Auslösen des Interrupt 5 wird der Bildschirminhalt gespeichert. Wenn das Betriebssystem dann Zeit hat (Interrupt 28hex) werden die Daten vom Bildschirm auf Diskette gesichert. Da beide Routinen auf Aktivität geprüft werden, dürfen Sie die Taste [PrtScrn] ruhig öfter betätigen, ohne daß das System abstürzt.

(Helmut Amshove/sw)

```
[VN4]    1: ;-----------------------------------------;
[L65]    2: ; E I N F A C H E  T S R - R O U T I N E ;
[1V6]    3: ;  Installation "zwischen" dem Interrupt ;
[1N4]    4: ;----------------------------------------- ;
[5Z1]    5: .model tiny
[FTØ]    6: .code
[A61]    7:          ORG 1ØØH
[817]    8: START: JMP INSTALL ;bei Aufruf installiern
[ØØØ]    9:
[ZK5]   1Ø: OLDINT   DD  Ø          ; alter Interrupt
[3Z1]   11: OLDSS    DW  Ø
[XY1]   12: OLDSP    DW  Ø
[D72]   13: LAENGE   DB  19h
[WU6]   14: Meldung  DB  'Hotkey: Taste F1 betätigt.'
[VN4]   15: ;-----------------------------------------;
[5E4]   16: ;   N e u e r   I N T 16                  ;
[VN4]   17: ;-----------------------------------------;
[BM4]   18: NEUINT: CMP AH,Ø        ; Taste?
[8PØ]   19:         JE OK
[1W3]   2Ø:         JMP CS:[OLDINT] ;nein,
[WV2]   21: OK:     PUSHF           ;
[US4]   22:         CALL CS:[OLDINT];Unterprog
[BE3]   23:         CMP AH,3BH      ;"Hotkey"
[TS3]   24:         JNE ENDE        ;nein, Ende
[NG3]   25:         CLI             ;ab hier die
[ØG4]   26:         MOV CS:OLDSS,SS ;Funktion
[8N5]   27:         MOV CS:OLDSP,SP ;Stack umstellen
[WT3]   28:         PUSH CS         ;auf cs:1ØØ
[7P4]   29:         POP  SS         ;(...auch machbar)
```

```
[6W1]   3Ø:         MOV SP,1ØØH
[PØ5]   31:         PUSH AX         ;Register sichern
[WC1]   32:         PUSH CX
[3B1]   33:         PUSH DI
[7X4]   34:         MOV  CH,Ø       ;Meldung ausgeben
[PR2]   35:         MOV CL,CS:LAENGE
[2A1]   36:         MOV DI,Ø
[2D1]   37:         MOV AH,14
[MG4]   38:  AUS:   MOV AL,CS:MELDUNG[DI]
[Y71]   39:         INT 1ØH
[BØ1]   4Ø:         INC DI
[VD1]   41:         LOOP AUS
[G21]   42:         POP DI
[V31]   43:         POP CX
[R31]   44:         POP AX
[EMØ]   45:         CLI
[UT2]   46:         MOV SS,CS:OLDSS
[5T2]   47:         MOV SP,CS:OLDSP
[6PØ]   48:         STI
[DT1]   49: ENDE:   IRET
[XV4]   5Ø: ;-----------------------------------------;
[YL6]   51: ; Installationsteil   TSR-Routine         ;
[XV4]   52: ;-----------------------------------------;
[SB7]   53: INSTALL:MOV AH,35H  ; alten Interrupt lesen
[AM1]   54:         MOV AL,16H
[381]   55:         INT 21H
[BU4]   56:         MOV  WORD PTR CS:[OLDINT],BX
[W55]   57:         MOV  WORD PTR CS:[OLDINT+2],ES
[8C1]   58:         PUSH CS
```

Listing 1. »int16.asm« ist ein einfaches Hotkey-Programm

```
[E31]   59:          POP DS
[7M5]   6Ø:          MOV DX, OFFSET NEUINT ; Setzen neu
[XL1]   61:          MOV AH,25H
[AM1]   62:          MOV AL,16H
[381]   63:          INT 21H
[XG5]   64:          LEA  DX,INSTALL ; Programmlänge
[L45]   65:          INT  27H     ; Beenden und resident
[W53]   66:          END START    ; bleiben
```

Listing 1. »int16.asm« ist ein einfaches Hotkey-Programm

```
[WA4]    1: ; ------------------------------------
[716]    2: ; Print Screen in Datei (Textschirm)
[P3Ø]    3: ;
[XC2]    4: ;  V e r s i o n     1
[WA4]    5: ; ------------------------------------
[Ø85]    6: .model Tiny        ; Speichermodell
[PK4]    7: .code              ; für COM Datei
[ØØØ]    8:
[8F2]    9: pushall  macro
[EW3]   1Ø:         push ax           ; Register
[BK3]   11:         push bx           ; sichern
[WC1]   12:         push cx
[ZC1]   13:         push dx
[BC1]   14:         push ds
[EC1]   15:         push es
[3B1]   16:         push di
[GC1]   17:         push si
[B25]   18:         push cs           ; und Datensegment
[D83]   19:         pop ds            ; setzen
[HPØ]   2Ø:         endm
[ØØØ]   21:
[992]   22: popall   macro
[UL3]   23:         pop si            ; Register
[794]   24:         pop di            ; restaurieren
[G31]   25:         pop es
[E31]   26:         pop ds
[X31]   27:         pop dx
[V31]   28:         pop cx
[T31]   29:         pop bx
[R31]   3Ø:         pop ax
[HPØ]   31:         endm
[ØØØ]   32:
[WA4]   33: ; ------------------------------------
[A61]   34:          org 1ØØh
[526]   35: start:   jmp install     ;programm inst
[WA4]   36: ; ------------------------------------
[LA2]   37: ; Datenbereich
[764]   38:          dw 1Ø dup (?)  ;eigener Stack
[542]   39: pstack    equ $
[ØØØ]   4Ø:
[KØ3]   41: Bildschirm dw Øb8ØØh
[ØØØ]   42:
[MF5]   43: filename db 'Bild'      ;Dateiname der
[434]   44: nr1      db 'Ø'         ;Bilddatei
[8V1]   45: nr2      db 'Ø'
[CV1]   46: nr3      db 'Ø'
[FA1]   47:          db '.txt'
[YFØ]   48:          db Ø
[ØØØ]   49:
[R43]   5Ø: ; Bildschirmpuffer
[M23]   51: dta      db 8Ø*24 dup (?)
[ØØØ]   52:
[AS4]   53: OldSS    dw Ø           ; Alter Stack
[2P1]   54: OldSP    dw Ø
[NC4]   55: aktiv    db Ø           ; Aktiv Flag
[ØØØ]   56:
[9C4]   57: ; ------------------------------------
[UF1]   58: ; I N T   5
[9C4]   59: ; ------------------------------------
[8C1]   6Ø: Neuint:
[MT5]   61:          cmp cs:[aktiv],Ø ; Interrupt aktiv
[W61]   62:          je lies
[BF4]   63:          iret              ; ja also zurück
[ØØØ]   64:
[RL3]   65: lies:    inc cs:[aktiv]
[EMØ]   66:          cli
[FØ6]   67:          mov cs:oldss,ss  ;Stack neu setzen
[6U2]   68:          mov cs:oldsp,sp
[6PØ]   69:          sti
[8C1]   7Ø:          push cs
[C41]   71:          pop ss
[ØF3]   72:          mov sp,offset pstack
[9E1]   73:          pushall
[LR4]   74:          inc nr3           ; Bildnr erhöhen
[CR1]   75:          cmp nr3,'9'
[F71]   76:          jle ok1
[3S1]   77:          mov nr3,'Ø'
[281]   78:          inc nr2
[8R1]   79:          cmp nr2,'9'
[F71]   8Ø:          jle ok1
```

```
[ZR1]   81:          mov nr2,'Ø'
[Y71]   82:          inc nr1
[D15]   83:    ok1:  mov cx,8Ø*24     ;8Øx24 Zeichen
[ULØ]   84:          cld
[BC1]   85:          push ds
[9N5]   86:          mov ax,[Bildschirm] ;von Schirm
[1H1]   87:          mov ds,ax
[ØB1]   88:          mov si,Ø
[8C1]   89:          push cs
[G31]   9Ø:          pop es
[RU3]   91:          lea di,dta         ;in Puffer
[S42]   92: lesen:   movsb
[911]   93:          inc si
[GS1]   94:          loop lesen
[E31]   95:          pop ds
[1M2]   96: ; ---------------------
[Ø66]   97: schreib1: lea  dx,filename ;Zeiger Name
[2K4]   98:           mov cx,2Øh       ;Fileattribut
[834]   99:           mov ah,3ch       ; neue Datei
[381]  1ØØ:           int 21h
[GT4]  1Ø1:           mov bx,ax        ;Handle sichern
[KT4]  1Ø2:           lea dx,dta       ;Pufferadresse
[X74]  1Ø3:           mov cx,8Ø*24     ;Anzahl Byte
[NL1]  1Ø4:           mov ah,4Øh
[PL4]  1Ø5:           int 21h     ; 8Ø*24 bytes in File
[YM1]  1Ø6:           mov ah,3eh
[Z84]  1Ø7:           int 21h            ;Close handle
[A41]  1Ø8:          popall
[EMØ]  1Ø9:          cli
[LB5]  11Ø:          mov ss,cs:oldss      ; alten Stack
[WH5]  111:          mov sp,cs:oldsp     ; restaurieren
[6PØ]  112:          sti
[LX4]  113:          dec cs:[aktiv]       ; aktiv flag
[ØN3]  114:          iret                 ; löschen
[WA4]  115: ; ------------------------------------
[NZ3]  116: ; Installationsroutine
[WA4]  117: ; ------------------------------------
[KS2]  118: install: push cs
[E31]  119:          pop ds
[HH3]  12Ø:          mov dx, offset Neuint
[465]  121:          mov ah,25h  ;Interrupt 5 setzen
[AA1]  122:          mov al,5
[381]  123:          int 21h
[PD2]  124:          lea dx,install
[2E4]  125:          int 27h     ;resident beenden
[ØØØ]  126:
[4K1]  127:      end start
```

Listing 2. Version 1 des Hardcopy-Programms: »tsr.asm«

```
[323]    1: ; -------------------------
[AE4]    2: ; Bildschirmtyp ermitteln
[323]    3: ; -------------------------
[6Y1]    4: Bildadr dw Ø
[EK4]    5: Bildspeicher db 8Ø*24*2 dup(?)
[ØØØ]    6:
[V72]    7: bildtyp proc
[XM1]    8:          mov ah,ØFh
[Y71]    9:          int 1Øh
[U81]   1Ø:          cmp al,7
[M71]   11:          je mono
[2Ø3]   12:          mov Bildadr,Øb8ØØh
[712]   13:          jmp endbild
[434]   14: mono:    mov Bildadr,ØbØØØh
[412]   15: endbild: ret
[MPØ]   16:          endp
[ØØØ]   17:
[PØ3]   18: ; -------------------------
[FD3]   19: ; Bildschirm sichern
[PØ3]   2Ø: ; -------------------------
[N72]   21: bildsich proc
[N22]   22:          mov cx,2*8Ø*24
[ULØ]   23:          cld
[BC1]   24:          push ds
[UU2]   25:          mov ax,[bildadr]
[1H1]   26:          mov ds,ax
[ØB1]   27:          mov si,Ø
[8C1]   28:          push cs
[G31]   29:          pop es
[P93]   3Ø:          lea di,bildspeicher
[522]   31:          repnz movsb
[E31]   32:          pop ds
[3PØ]   33:          ret
[MPØ]   34:          endp
[ØØØ]   35:
[2V2]   36: ; ------------------------
[KA4]   37: ; Bildschirm restaurieren
[4U2]   38: , ------------------------
[L92]   39: bildrest proc
```

```
[N22]   4Ø:          mov cx,2*8Ø*24
[ULØ]   41:          cld
[BC1]   42:          push ds
[UU2]   43:          mov ax,[bildadr]
[3H1]   44:          mov es,ax
[2A1]   45:          mov di,Ø
[8C1]   46:          push cs
[E31]   47:          pop ds
[MA3]   48:          lea si,bildspeicher
[522]   49:          repnz movsb
[E31]   5Ø:          pop ds
[3PØ]   51:          ret
[MPØ]   52:          endp
```

Listing 3. Mit »bildsch.asm« wird der Grafikkartentyp ermittelt

```
[WA4]    1: ; ---------------------------------------
[T16]    2: ; Print Screen in Datei (Textschirm)
[P3Ø]    3: ;
[ØD2]    4: ;  V e r s i o n     2
[WA4]    5: ; ---------------------------------------
[Ø85]    6: .model Tiny        ; Speichermodell
[PK4]    7: .code              ; für COM Datei
[ØØØ]    8:
[8F2]    9: pushall  macro
[EW3]   1Ø:         push ax          ; Register
[BK3]   11:         push bx          ; sichern
[WC1]   12:         push cx
[ZC1]   13:         push dx
[BC1]   14:         push ds
[EC1]   15:         push es
[3B1]   16:         push di
[GC1]   17:         push si
[B25]   18:         push cs          ; und Datensegment
[D83]   19:         pop ds           ; setzen
[HPØ]   2Ø:         endm
[ØØØ]   21:
[992]   22: popall   macro
[UL3]   23:         pop si           ; Register
[794]   24:         pop di           ; restaurieren
[G31]   25:         pop es
[E31]   26:         pop ds
[X31]   27:         pop dx
[V31]   28:         pop cx
[T31]   29:         pop bx
[R31]   3Ø:         pop ax
[HPØ]   31:         endm
[ØØØ]   32:
[WA4]   33: ; ---------------------------------------
[A61]   34:          org 1ØØh
[9N6]   35: start:  jmp install     ;Programminstall
[WA4]   36: ; ---------------------------------------
[LA2]   37: ; Datenbereich
[764]   38:          dw 1Ø dup (?)  ;eigener Stack
[542]   39: pstack   equ $
[ØØØ]   4Ø:
[KØ3]   41: Bildschirm dw Øb8ØØh
[ØØØ]   42:
[MF5]   43: filename db 'Bild'      ;Dateiname der
[434]   44: nr1      db 'Ø'         ;Bilddatei
[8V1]   45: nr2      db 'Ø'
[CV1]   46: nr3      db 'Ø'
[FA1]   47:          db '.txt'
[YFØ]   48:          db Ø
[ØØØ]   49:
[R43]   5Ø: ; Bildschirmpuffer
[M23]   51: dta      db 8Ø*24 dup (?)
[ØØØ]   52:
[AS4]   53: OldSS    dw Ø           ; Alter Stack
[2P1]   54: OldSP    dw Ø
[WD4]   55: aktiv    db Ø           ; Aktiv-Flag
[Ø75]   56: ok       db Ø           ; Bild zum Sichern
[V22]   57: Altint28  dd Ø
[ØØØ]   58:
[3C4]   59: ;---------------------------------------
[9F5]   6Ø: Neuint28: pushf                ;alten Int
[5Z4]   61:           call cs:[altint28]  ;aufrufen
[L15]   62:           cmp cs:[ok],Ø     ; Bild sichern?
[ØØ2]   63:           jne schreib
[814]   64:           iret              ; nein zurück
[US6]   65: schreib:  cmp cs:[aktiv],Ø ; bereits aktiv
[XT1]   66:           je schreib1
[WA3]   67:           iret              ; zurück
[1M2]   68: ; ---------------------
[157]   69: schreib1: inc cs:[aktiv] ;Aktivflag setzen
[EMØ]   7Ø:           cli
```

```
[WP5]   71:           mov cs:oldss,ss ;Stack neusetzen
[6U2]   72:           mov cs:oldsp,sp
[6PØ]   73:           sti
[8C1]   74:           push cs
[C41]   75:           pop ss
[ØF3]   76:           mov sp,offset pstack
[9E1]   77:           pushall
[EG5]   78:           lea  dx,filename;Zeiger auf Name
[YP4]   79:           mov cx,2Øh          ;Fileattribut
[Y94]   8Ø:           mov ah,3ch          ; neue Datei
[381]   81:           int 21h
[GT4]   82:           mov bx,ax         ;Handle sichern
[CR4]   83:           lea dx,dta         ;Pufferadresse
[RF4]   84:           mov cx,8Ø*24        ;Anzahl Byte
[NL1]   85:           mov ah,4Øh
[FG4]   86:           int 21h     ; 8Ø*24 Byte in Datei
[YM1]   87:           mov ah,3eh
[Z84]   88:           int 21h             ;Close handle
[RX3]   89:           dec ok              ;Ok zurück
[VØ5]   9Ø:           popall            ;Register zurück
[8L5]   91:           dec cs:[aktiv]   ;Aktiv löschen
[EMØ]   92:           cli
[UG5]   93:           mov ss,cs:oldss     ;Stack zurück
[5T2]   94:           mov sp,cs:oldsp
[6PØ]   95:           sti
[MP2]   96:           iret                ;Ende
[ØØØ]   97:
[9C4]   98: ; ---------------------------------------
[UF1]   99: ; I N T   5
[9C4]  1ØØ: ; ---------------------------------------
[8C1]  1Ø1: Neuint:
[MT5]  1Ø2:          cmp cs:[aktiv],Ø ; Interrupt aktiv
[W61]  1Ø3:          je lies
[BF4]  1Ø4:          iret              ; ja also zurück
[ØØØ]  1Ø5:
[RL3]  1Ø6: lies:    inc cs:[aktiv]
[EMØ]  1Ø7:          cli
[FØ6]  1Ø8:          mov cs:oldss,ss  ;Stack neu setzen
[6U2]  1Ø9:          mov cs:oldsp,sp
[6PØ]  11Ø:          sti
[8C1]  111:          push cs
[C41]  112:          pop ss
[ØF3]  113:          mov sp,offset pstack
[9E1]  114:          pushall
[LR4]  115:          inc nr3           ; Bildnr erhöhen
[CR1]  116:          cmp nr3,'9'
[F71]  117:          jle ok1
[3S1]  118:          mov nr3,'Ø'
[281]  119:          inc nr2
[8R1]  12Ø:          cmp nr2,'9'
[F71]  121:          jle ok1
[ZR1]  122:          mov nr2,'Ø'
[Y71]  123:          inc nr1
[955]  124:    ok1:  mov cx,8Ø*24     ; 8Øx24 Zeichen
[ULØ]  125:          cld
[BC1]  126:          push ds
[Ø56]  127:          mov ax,[Bildschirm];von Bildschirm
[1H1]  128:          mov ds,ax
[ØB1]  129:          mov si,Ø
[8C1]  13Ø:          push cs
[G31]  131:          pop es
[2Z3]  132:          lea di,dta       ; in Puffer
[S42]  133: lesen:   movsb
[911]  134:          inc si
[GS1]  135:          loop lesen
[E31]  136:          pop ds
[711]  137:          inc ok
[BH1]  138:          dec aktiv
[A41]  139:          popall
[UT2]  14Ø:          mov ss,cs:oldss
[5T2]  141:          mov sp,cs:oldsp
[GRØ]  142:          iret
[WA4]  143: ; ---------------------------------------
[NZ3]  144: ; Installationsroutine
[WA4]  145: ; ---------------------------------------
[KS2]  146: install: push cs
[E31]  147:          pop ds
[S25]  148:          mov ah,35h  ;Interrupt 28 lesen
[FM1]  149:          mov al,28h
[381]  15Ø:          int 21h
[B55]  151:          mov word ptr cs:[altint28],bx
[BG5]  152:          mov word ptr cs:[altint28+2],es
[HH3]  153:          mov dx, offset Neuint
[L75]  154:          mov ah,25h  ; Interrupt 5 setzen
[AA1]  155:          mov al,5
[381]  156:          int 21h
[ZS3]  157:          mov dx,offset Neuint28
[E95]  158:          mov ah,25h  ; Interrupt 28 setzen
[FM1]  159:          mov al,28h
[381]  16Ø:          int 21h
[PD2]  161:          lea dx,install
[2E4]  162:          int 27h      ;resident beenden
[ØØØ]  163:
[4K1]  164:      end start
```

Listing 4. Jetzt perfekt: »tsr2.asm« speichert Ihre Hardcopies

Des Assemblers neue Kleider

Die Version 4.0 des Microsoft Makroassemblers (MASM) wurde vom Hersteller bereits vor einiger Zeit grundlegend überarbeitet. Das Ergebnis war die Version 5.0, auf die relativ schnell die Version 5.1 folgte. Markanteste Merkmale der neuen Versionen sind vereinfachte Segmentdefinitionen, ein verbessertes Hochsprachen-Interface sowie die Tatsache, daß der MASM 5.1 auch in einer OS/2-Version existiert.

Wohl bei keinem anderen Softwareprodukt dürften die Ansprüche, die der Benutzer an das Programm stellt, so stark variieren wie bei einem Assembler. Die einen erwarten lediglich ein Programm, das in der Lage sein soll, die Mnemonics (Befehlskürzel des Assemblers) in die entsprechenden Opcodes (ausführbarer Maschinenbefehl in Form eines Bytes oder einer Folge von Bytes) umzuwandeln. Die anderen wünschen eine komfortable, nach Möglichkeit fensterorientierte Benutzeroberfläche und vor allem leistungsfähige Anweisungen, die dem Benutzer möglichst viel Arbeit abnehmen sollen.
Wenn der Makroassembler bis zur Version 4.0 eher den kargen Ansprüchen der ersten Gruppe genügte, hat sich der Bedienungskomfort des Assemblers ab der Version 5.0 erheblich erhöht. Zwar bietet MASM (noch) keine Fensteroberfläche à la Turbo Pascal, doch wird durch die neuen Anweisungen der notwendige Formalismus beim Erzeugen eines Assemblerprogramms auf ein Minimum reduziert, vor allem, wenn dieses Modul von einem Hochsprachenprogramm aufgerufen werden soll. Insbesondere Einsteiger, die um die Assemblerprogrammierung bislang eher einen großen Bogen machten, profitieren von den neuen Anweisungen.
Die Tabelle gibt Ihnen zunächst eine Übersicht über die Neuerungen in den Versionen 5.0 beziehungsweise 5.1. Im folgenden werden alle vorgestellten Änderungen auf die Version 5.1 bezogen, da wohl niemand auf die Version 5.0 umsteigen wird, wenn es bereits eine Nachfolgeversion gibt.
Aus Platzgründen lassen sich leider nicht alle Neuerungen ausführlich vorstellen. In dieser Übersicht geht es daher in erster Linie um die beiden wichtigsten Verbesserungen – die vereinfachten Segmentdefinitionen und das Hochsprachen-Interface, das heißt die Schnittstelle zwischen Assembler und Hochsprachen wie C oder Pascal.

Was es mit den Segmentanweisungen im allgemeinen auf sich hat, wurde bereits an anderer Stelle dieser DOS-Extra-Ausgabe ausführlich erläutert. Trotzdem soll nochmals kurz auf die Unterschiede zwischen der Version 4.0 und 5.1 eingegangen werden. Bis zur Version 4.0 des MASM ließen sich Segmente ausschließlich über die Segmentanweisung definieren:

```
test segment
....
test ends
```

Die Definition des obigen Segments mit dem Namen »test« wird durch die Anweisung »test segment« eingeleitet und durch die Anweisung »test ends« wieder beendet. Diese Anweisung – die in der Version 5.1 nach wie vor zur Verfügung steht – erlaubt es, den Segmentnamen sowie einige zusätzliche Segmentparameter, die die spätere Lage des Segments im Arbeitsspeicher bestimmen, frei zu wählen. Eindeutiger Nachteil dieser Anweisung ist jedoch eine umständliche Handhabung, insbesondere für Einsteiger. Da in den meisten Fällen die Optionen der Segmentanweisung ohnehin nicht genutzt werden, haben sich die Entwickler bei Microsoft zu einigen drastischen Vereinfachungen entschlossen.
In der neuen Version definieren Sie Segmente auch ohne die Segmentanweisung mit Hilfe der vereinfachten Anweisungen (auch die leidige Assume-Anweisung ist verschwunden). So wird beispielsweise das Stack-Segment durch ».Stack«, das Datensegment durch ».Data« und das Programmsegment durch ».Code« definiert.

```
a) MASM 4.0

_DATA     SEGMENT WORD PUBLIC 'DATA'
_DATA     ENDS
DGROUP    GROUP _DATA

_TEXT     SEGMENT WORD PUBLIC 'CODE'
          ASSUME CS:_TEXT,DS:DGROUP,SS:DGROUP
          PUBLIC _Addiere
_Addiere  PROC  NEAR
          PUSH BP
          MOV BP,SP
          SUB SP,2
          PUSH CX
          MOV WORD PTR[BP-2],0
          MOV CX,WORD PTR[BP+6]
LAB1:     ADD AX,[BP+4]
          LOOP LAB1
          POP CX
          MOV SP,BP
          POP BP
          RET
_Addiere  ENDP
_TEXT     ENDS
          END



b) MASM 5.1

.MODEL SMALL,C
.CODE
Addiere PROC USES CX,ZAHL:WORD,ANZAHL:WORD
              LOCAL SUMME
              MOV SUMME,0
              MOV CX,ANZAHL
LAB1:         ADD AX,ZAHL
              LOOP LAB1
              RET
Addiere       ENDP
              END
```

Bild 1. Ein Assemblerprogramm in der Version 4.0 und 5.1 des Makroassemblers

Für Einsteiger empfehlenswert: die vereinfachten Segmentanweisungen

Da den Segmenten keine Namen gegeben werden, stehen zusätzlich Textkonstanten zur Verfügung, um die einzelnen Segmente innerhalb des Programms auch ansprechen zu können. Für das Programmsegment ist die Textkonstante »@Code«, für das Datensegment die Textkonstante »@Data« zuständig. Wie sich durch die neuen Anweisungen der Aufbau eines Assemblerprogramms vereinfacht, zeigt Bild 1. Bei diesem Beispiel fällt zum einen auf, daß der Quelltext in der Version 5.1 wesentlich kürzer geworden ist, und zum anderen, daß Stack-Parameter in der neuen Version über einen symbolischen Namen angesprochen werden. Lassen Sie sich durch den optischen Eindruck aber nicht täuschen. Es reduziert sich lediglich der Aufwand für das Schreiben des Quelltextes – das assemblierte Maschinenprogramm ist in beiden Versionen identisch (Sie können das Beispielprogramm jedoch nicht ohne weiteres aufrufen, da es für den Einsatz innerhalb eines C-Programms konzipiert ist).
Um die vereinfachten Segmentanweisungen zu nutzen, müssen Sie allerdings ein Speichermodell mit der Anweisung ».Model« vereinbaren. Das Speichermodell (zur Auswahl stehen »Small«, »Compact«, »Medium« und »Large«) legt fest, ob Codebeziehungsweise Datenbereiche kleiner oder größer als 64 KByte (der maximalen Größe eines Segments) werden dürfen. Diese Angabe benötigt der Assembler, um bei Segmentdefinitionen die Segmentnamen beziehungsweise Segmentparameter und

bei Prozeduren den korrekten Rückkehrbefehl (»retn« oder »retf«) zu erzeugen, denn durch die vereinfachten Segmentanweisungen steht ihm die entsprechende Information nicht mehr zur Verfügung.
Doch die vereinfachten Segmentanweisungen bringen nicht nur eine Erleichterung bei der Definition von Segmenten. Ihr besonderer Vorteil liegt darin, daß der Assembler Segmentnamen und Segmentparameter so erzeugt, daß sie mit jenen übereinstimmen, die ein Microsoft-Compiler erzeugt. Dies ist nämlich eine der wichtigsten Voraussetzungen, um Hochsprachenprogramme mit Assemblerprogrammen kombinieren zu können.
Heutzutage werden in immer geringerem Maße große Applikationen ausschließlich in Assembler geschrieben. Dank leistungsfähiger optimierender Compiler läßt das Laufzeitverhalten von compilierten C- oder Modula-2-Programmen selten Wünsche offen. Sollte die Ausführungsgeschwindigkeit dennoch nicht ausreichen, ist es wesentlich sinnvoller (weil ökonomischer), Anwendungen in einer Hochsprache zu schreiben und nur jene Teile des Programms, die entweder sehr zeitkritisch sind oder direkt auf die Systemhardware zugreifen müssen, in Maschinensprache zu programmieren.

Der MASM 5.1 besitzt ein effektives Hochsprachen-Interface

In der Version 4.0 des Makroassemblers mußte sich der Programmierer noch um jede Kleinigkeit selbst kümmern – ein Umstand, der die Assemblerprogrammierung nicht sehr attraktiv machte. Um dem Hochsprachenprogrammierer die Einbindung von Maschinenroutinen zu erleichtern, stellt die neue MASM-Version ein einfaches, aber sehr effektives Hochsprachen-Interface zur Verfügung.
Unter dem Begriff Hochsprachen-Interface sind eine Reihe von Konventionen zusammengefaßt, die erfüllt sein müssen, wenn von einem Hochsprachenprogramm ein Maschinenprogramm (oder auch umgekehrt) aufgerufen werden soll. Gemäß der ursprünglichen Bedeutung dieses Begriffs gibt es für alle sogenannten »gemischtsprachigen« Programme eine derartige Konvention, also beispielsweise auch für den Aufruf von Hochsprachenprogrammen von einer anderen Hochsprache aus. Darüber hinaus zählen zum Hochsprachen-Interface auch jene Anweisungen, die den Aufruf einer Assemblerroutine von einem Hochsprachenprogramm erlauben.

```
.MODEL SMALL,C
.CODE
  SORTIERE PROC USES
DI,SI,LAENGE:WORD,ZEIGER:PTR
            ...
            Hier folgt die
            eigentliche Prozedur
            ...
            RET
  SORTIERE ENDP
END
```

Bild 2 Das Assemblermodul »sortiere«

Stellen Sie sich vor, Sie wollen aus einem C-Programm eine Maschinenroutine aufrufen. Für das C-Programm ergeben sich keine besonderen Konsequenzen, denn der Aufruf der Maschinenroutine entspricht der einer externen C-Funktion. Zunächst deklarieren Sie die Funktion

```
extern int sortiere(int laenge, int feld[
]);
```

die Sie dann innerhalb des C-Programms wie eine normale C-Funktion aufrufen:

```
sortiere(laenge, feld);
```

Im Assemblerprogramm muß nun eine Prozedur mit dem Namen »sortiere« definiert werden (Bild 2).
Wenn Sie sich dieses Bild einmal genauer betrachten, fällt als erstes die erweiterte Model-Anweisung auf. Durch den zusätzlichen Parameter »C« werden die Namens-, Parameter- und Aufrufkonventionen so vereinbart, wie sie das C-Hauptprogramm erwartet. Damit entfällt zum Beispiel die Notwendigkeit, den Prozedurnamen mit einem Unterstrich (»_«) beginnen oder als global deklarieren zu müssen, da dies der Assembler nun automatisch übernimmt.
Die eigentliche Prozedur wird wie üblich durch eine Proc-Anweisung eingeleitet. Allerdings wurde die Proc-Anweisung in der Version 5.1 des Makroassemblers erheblich erweitert. Zwar steht diese Anweisung in der Version 5.0 auch schon zur Verfügung, jedoch in Form eines Makros, das sich in der Datei »mixed.inc« befindet.
Die Anweisung »uses« legt Register fest, die während der Prozedur zwischengespeichert werden müssen. Der Assembler erzeugt für diese Register die benötigten Push-Befehle zum Sichern und am Ende der Prozedur die entsprechenden Pop-Befehle zum Wiederherstellen der Registerinhalte.
Als nächstes werden die Variablen festgelegt, die als Parameter beim Prozeduraufruf zu übergeben sind. Die C-Funktion legt nämlich die Werte dieser Parameter vor dem Funktionsaufruf auf dem Stack ab. Führen Sie diese Parameter auch innerhalb der Proc-Anweisung auf, dann erzeugt der Assembler Textkonstanten, mit deren Hilfe Sie Zugriff auf diese Parameter haben.

So laden Sie beispielsweise mit dem Befehl »mov cx,laenge« den übergebenen Wert der Variablen »laenge« in das CX-Register. Der Assembler hat für diese Variable das Textmakro »[bp + 6]« erzeugt, das den Wert auf dem Stack adressiert. Anstelle des Befehls »mov cx,laenge« wird also der Befehl »mov cx,[bp + 6]« assembliert.

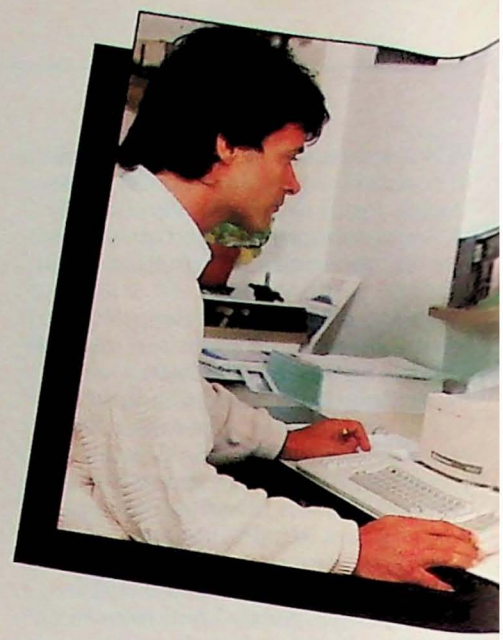
Vereinfachter Zugriff auf Variablen einer Hochsprache

Ein ähnliches Textmakro (allerdings mit einem negativen Offset) erzeugt die Anweisung »local«, durch die sich lokale Prozedurvariablen erzeugen lassen, die ebenfalls auf dem Stack abgelegt werden. Sie sehen an diesem Beispiel ganz gut, daß die neuen Anweisungen den Assembler nicht auf »geheimnisvolle« Weise erweitern; sie assemblieren vielmehr Befehle, um die sich der Programmierer in den vorhergehenden Versionen noch hatte selber kümmern müssen. Beendet wird die Prozedur durch den Befehl »ret«, der die Rückkehr zum C-Programm durchführt.
Sie sehen, daß die neuen Anweisungen – vor allem für den nicht so versierten Assemblerprogrammierer – eine enorme Arbeitserleichterung bedeuten. Es sei an dieser Stelle noch einmal darauf hingewiesen, daß sich das Einbinden von Assemblerroutinen in Hochsprachen natürlich auch mit älteren MASM-Versionen durchführen läßt. Aufgrund der neuen Anweisungen wird dem Programmierer aber viel Arbeit abgenommen, und vor allem wird die Anzahl der potentiellen Fehlerquellen drastisch reduziert.
Neben solchen eher spektakulären Erweiterungen, wie den neuen Segmentdefinitionen oder dem verbesserten Hochsprachen-Interface, wurden in der Version 5.1 aber auch zahlreiche kleinere Verbesserungen durchgeführt. Dazu gehören neue Optionen. Im Zusammenhang mit dem vorgestellten Beispiel ist die Option »/DSymbol« erwähnenswert. Durch sie wird ein Symbol beim Aufruf des Assemblers definiert. Sie könnten daher die erweiterte Model-Anweisung wie folgt deklarieren:

```
% .model small,sprache
```

Bei »sprache« handelt es sich um eine Variable, die den Sprachtyp festlegt. Der Operator »%« (steht ab der

Ausgefeilte Programmiertechnik und praxiserprobte Musterprogramme



inkl. aller 8086/8088 und 80186 Prozessorbefehle!

Praxiserprobte Musterprogramme in Assembler

Das Intel 16/32 Bit Assemblerhandbuch bietet Ihnen

- **systematische Programmierkurse** mit sofort nachvollziehbaren Übungsbeispielen, z. B. zur Stringverarbeitung, Interruptprogrammierung oder Makrotechnik, Assemblerkurs MASM 5.x und Code-View.
- **anschauliche Befehlsbeschreibungen** der Prozessoren 80(x)86 sowie der Coprozessoren 80(x)87. Zu jedem Befehl erhalten Sie eine detaillierte Funktionsbeschreibung, ein praktisches Anwendungsbeispiel, die Debugsitzung sowie Maschinencode und Zeitbedarf.
- **ausführliche Beschreibung von Architekturen und Registern** dieser Prozessoren.
- **Makrobibliotheken:** u. a. Stringbibliothek, ROM-BIOS-Routinen zur Druckeransteuerung.
- **Assemblerentwicklungssystem:** Von Anfang an sind Sie bei der Entwicklung eines Makroassemblers für den 8086-Prozessor dabei. Schritt für Schritt erhalten Sie ein exzellentes Programmierwerkzeug.
- **bewährte Musterlösungen in Assembler:** Sie erhalten **Systemprogramme**, u.a. Fenster für DOS, Ändern der Fatal-Error-Interrupt-Routine, **Druckerutilities** wie ein Druckertestprogramm, **Schnittstellenprogramme**, u. a. ein Programm zum Datentransfer zwischen PC-/MS-DOS-Rechnern, **technische Programme** wie das Muster einer Interruptroutine oder die Programmierung des timers 8253-5, **mathematische Programme**, u.a. CRC-Prüfsummen/-wörter für Dateien, **Anwenderprogramme** wie Passwörter für Systemdisketten oder Ver- und Entschlüsseln von EXE-Dateien sowie **Utilities** wie z.B. zum Dateien-Dump.
- **die Grundwerks-Diskette** mit sofort einsetzbaren Assemblerprogrammen und Utilities,
- **Ergänzungsausgaben zum Grundwerk** mit aktuellen Prozessorbeschreibungen (80386), neuen Musterprogrammen, nützlichen Routinen und Erweiterungen für die Makrobibliothek.

Fordern Sie noch heute an:

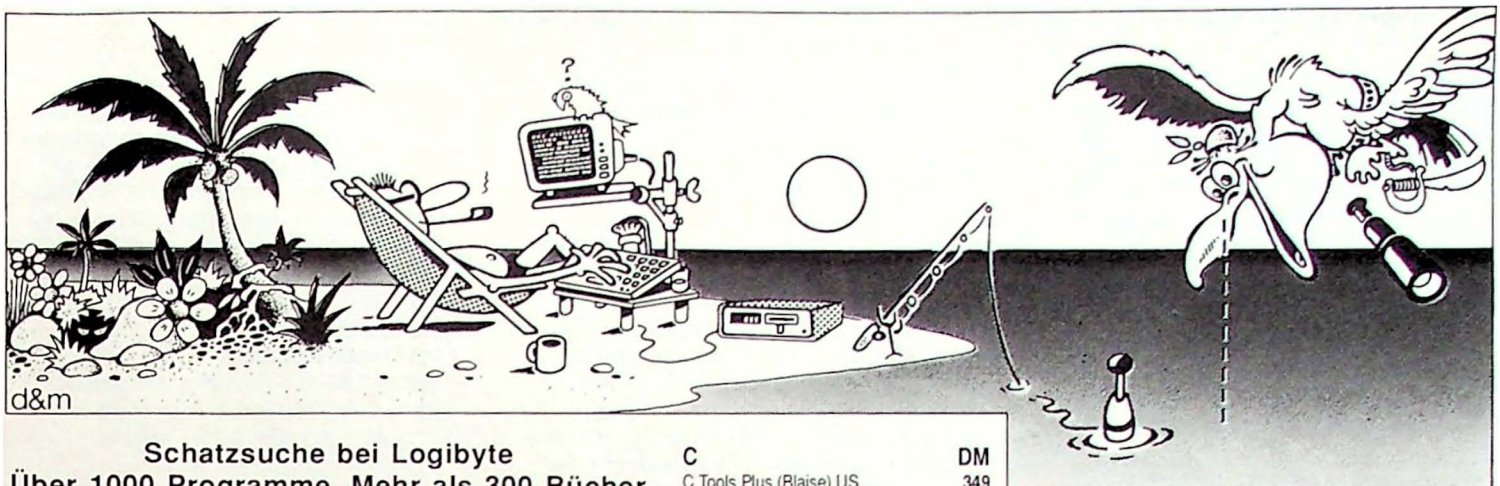
Intel 16/32 Bit Assemblerhandbuch

stabiler Ringbuchordner, DIN A4, ca. 1100 S., inkl. Diskette, Bestell-Nr.: 2200/72200, Preis: DM 98,–

Alle 2 – 3 Monate erhalten Sie Ergänzungsausgaben zum Grundwerk mit je ca. 130 Seiten zum Seitenpreis von 38 Pfennig (Abbestellung jederzeit möglich)

Für Ihre Anforderung verwenden Sie bitte nebenstehende Bestellkarte

INTEREST-VERLAG
Fachverlag für Special Interest Publikationen und Anwendersoftware
Industriestraße 21
D-8901 Kissing
Tel. 08233/2101-0



Schatzsuche bei Logibyte

Über 1000 Programme, Mehr als 300 Bücher

- Wir beraten Sie bei der Auswahl Ihres Programmes.
- Sie erhalten nur neueste Originalversionen.
- Bitte stets den aktuellen Gesamtkatalog anfordern.

Datenbanken	DM
Btrieve 5.x US	569
Clipper Sommer 87 US/dt.	1299/1599
David	439
dBase IV US/dt.	1469/1599
dBase IV Entwickler US/dt.	2199/2299
DBXL Diamond US/dt.	339/669
F & A 3.0 dt.	1049
Foxbase Plus 2.1 US/dt.	699/1449
Foxbase 286 Entwickler dt.	2399
Informix SQL 2.1 dt.	1699
Kartei Privat dt.	99
Oracle 5.1B	2499
Paradox 3.0 US/2.0 dt.	1599
R:Base 1.01 dt.	199
WordAdress III für MS-Word + WP	399

Grafik	DM
Arts & Letters Editor US	1299
Backchart 2000 mit GEM dt.	249
Designer 2.0 US/dt.	1469/1699
Freelance Plus 3.0 US/dt.	999/1199
GEM Artline US/dt.	899/1099
GEM Draw Plus dt.	539
GEM Presentation Team dt.	999
Graph Plus US/dt.	899/1199
Harvard Graphics 2.12 US/dt.	899/949
Hijack US	369
MS-Chart US/dt.	649/739
MS-Windows Draw US/dt.	319/429

CAD	DM
AutoCAD 9.0 Standard	1099
AutoCAD 10.0 US/dt.	6999/8999
Autoshade dt.	1899
AutoCAD - ADD ON's	Telefon
Autosketch 1.03	149
Autosketch 1.04 dt.	199
EasyCAD 2 US + MS-Maus/dt.	je 499
OrCAD STD III US/dt.Handb.	1599/1769
Generic CAD 3.0/3D US	299/599
Generic CADD Level 3 dt.	1699

Textverarbeitung	DM
Manuscript 2.0 US/dt.	969/999
MS-Pageview dt.	99
Starwriter PC 3.02 dt.	269
Starwriter 5.0 dt.	439
Textmaker 2.01 dt.	222
Witchpen Combi dt.	222
Word Junior dt.	269
Word 5.0 US/4.0 dt.	639/899
Word 4.0 mit Pageview dt.	969
Word mit WordAdress III dt.	1299
WordPerfect 5.0 US/dt.	639/969
Wordstar 5.0 Professional dt.	849

Integrierte Programme	DM
Enable 3.0 US/2.15 dt.	1199/1699
Framework III 1.1 dt.	1449
MS-Works US/dt.	329/399
Smartware 3.1 dt.	1899
Symphony 2.0 US/dt. *NEU*	Telefon

Desktop Publishing	DM
Bitstream-Fonts	je 499
First Publisher US/dt.	299/399
Gem Desktop Publisher dt.	899
Goscript/Plus US	499/999
Pagemaker 3.0 US/dt.	1399/1699
Timeworks Publisher mit GEM dt.	469
Ventura Publisher 2.0 US/dt.	1449/2099
Ventura Publ. Prof. Ext.	1169

Tabellenkalkulation	DM
Lotus 1-2-3 3.0 US/2.01 dt.	je 939
MS-Excel 2.1 US/dt.	799/999
MS-Multiplan 4.0 US/dt.	429/569
MS-Multiplan Junior dt.	169
Quattro US/dt.	449/469
Starplaner 2.0 dt.	349
Supercalc V US/dt.	769/1199

Mäuse	DM
Logi-Maus dt.	269
MS-Maus mit Paintbrush dt.	279
MS-Maus mit EasyCAD US/dt.	499/699

Utilities	DM
Allways dt.	379
Copy II PC 5.xx US	79
Desqview 2.2 US	299
Disk Optimizer 4.0 US	199
Faceguard dt.	249
Fastback Plus US/dt.	349/469
Formular Manager 3.0 dt.	369
Norton Commander 2.0 US/dt.	je 199
Norton Utilities 4.5 US/dt.	209/229
Norton Adv.Util. 4.5 US/dt.	je 279
PC Tools DeLuxe 5.1 US/5.5 US	159/259
PC Tools DeLuxe 4.3x dt.	149
Pizzaz Plus US/dt.	229/329
Savedir dt.	179
Say What dt.	135
Sidekick Plus dt.	349
Sideways 3.12 US/dt.	169/219
Snap dt.	99
Speedstore 5.1x	99
Star Manager 2.0 dt.	89
Turbo Backup 5.0 dt.	399
Windows 286 US/dt.	229/319
Windows 386 US/dt.	399/499
WP Library 2.0 dt.	299
XTree US/dt.	159/199
XTree Pro US/dt.	239/269

Assembler/Disassembler	DM
MS-Macro Assembler 5.1 US	269
Sourcer mit Bios Preprocessor US	379
Turbo Assembler/Debugger dt.	299

Basic	DM
Graph Pack für Turbo Basic US	189
MS-Quick Basic 4.5 US/dt.	199/239
QuickPak Professional mit dt. HB	389
Turbo Basic dt.	229
Turbo Basic Toolbox dt.	je 199

C	DM
C Tools Plus (Blaise) US	349
MS-C Compiler 5.1 US	899
MS-C Library Source	229
MS-Quick C 2.0 US/dt.	199/239
Power C mit dt. Handbuch	129
Turbo C 2.0 dt.	299
Turbo C 2.0 Professional dt.	469
Turbo C Library Source US	399
Turbo C Tools (Blaise) US	349
Vitamin C US	499
über 100 C Programme siehe Katalog	

Cobol	DM
MS-Cobol 3.0 US	1599
Personal Cobol	499
RM/Cobol-85 2.01	2299

Editoren	DM
Brief 2.1 mit dt. Zeichensatz	599
KEdit 4.0 US	429
Norton Editor 1.3 US/dt.	199/209
SPF/PC mit dt. Handbuch	599
Vedit Plus US	379

Fortran	DM
Lahey Fortran 77L US	1299
Fortran QT Tools Library	799
MS-Fortran 5.0 US	799
RM/Fortran mit Forte 2.43 US	1299

Modula	DM
JPI TopSpeed Modula 2 dt.	279
JPI TopSpeed Modula Techkit US	179
JPI TopSpeed Modula Debugger dt.	179
JPI TopSpeed Gesamtpaket (3)	555

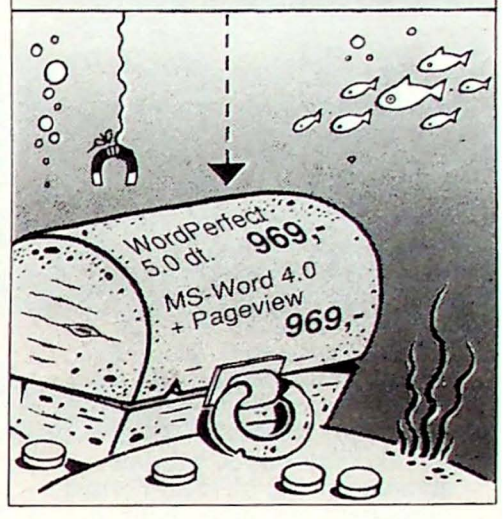
Pascal	DM
MS-Quick Pascal 1.0 US/dt.	199/239
Turbo Analyst 5.03 dt.	239
Turbo Pascal 5.5 US/5.0 dt.	je 299
Turbo Pascal 5.0 Professional dt.	469
Turbo Professional 5.06 dt.	269
Turbo Toolboxen dt.	je 199

Unix/Xenix	DM
SCO Xenix 286 System V	1499
SCO Xenix 286 System V komplett	3299
SCO Xenix 386 System V	1799
SCO Xenix 386 System V komplett	3699
VP/IX Dos-Progr. unter Unix	1399

HITS * HITS * HITS * HITS * HITS	
Carbon Copy Plus 5.1 US	369
Flugsimulator 3.0 dt.	139
KHK Auftragsbearbeitung dt.	350
KHK Finanzbuchhaltung dt.	350
MS-Dos Lernprogramm 2.0	89
NCSS, Statistik und Grafik	699
PC Scheme 3.03 US	299
Smalltalk/V	299
Smalltalk/V 286	469
Smalltalk Goodies	je 169
SPSS/PC + 3.0 US mit dt. Instal.	2269
Superprojekt Expert US/dt.	1499/2499
Turbo Jumbo Pack	599
Turbo Prolog 2.0 US/dt.	je 299
Zortech C++ US	279

Programmiersprachen/Tools/ Andere

Ada • APL • Betriebssystem • Betriebswirtschaft • Cross Tools • dBase Tools Debugger • Expertensysteme • Forth Kommunikation • Linker • Lisp Programmgeneratoren • Projektmanagment • Prolog • Schulversionen Statistik/Mathematik • Updates
Gratis-Gesamtübersicht anfordern !!!



Wir suchen:

- Sammelbesteller — Unterlagen bitte schriftlich anfordern.
- Softwarehäuser mit neuen interessanten Produkten zum Vertrieb.

Versand: Lieferung per Vorkasse ohne zusätzliche Versandkosten. Ihren V-Scheck lösen wir frühestens am Versandtag ein. Lieferung per Nachnahme zzgl. anfallender Versandkosten (nach Gewicht). Behörden, Universitäten und Großfirmen werden auf Rechnung beliefert. Lieferung ins Ausland nur per Nachnahme oder Vorkasse.
Alle Preise freibleibend.

Öffnungszeit Mo.-Fr. von 10-17 Uhr.

Bitte benutzen Sie zur Anforderung der Gesamtunterlagen die beigefügte Karte

Pfannstiel's **Logibyte**
SOFTWARE & BÜCHER

Kurfürstendamm 100
1000 Berlin 31
Tel. (030) 3 24 30 61
Fax. (030) 3 24 81 45

Version 5.1 zur Verfügung) bewirkt, daß MASM auch das Operandenfeld nach Textmakros (in diesem Fall »sprache«) untersucht. Über die Option »/D« bestimmen Sie nun beim Aufruf des Assemblers den Sprachtyp:

```
masm test /Dsprache = pascal;
```

Damit läßt sich eine allgemeine Prozedur erzeugen, die erst beim Assemblieren an die Koventionen der gewählten Sprache (in diesem Fall Pascal) angepaßt wird. In Abhängigkeit von der angegebenen Sprache erzeugt der Assembler nun die erforderlichen Parameter für das Hochsprachen-Interface.

Auch das Arbeiten mit Prozdeduren innerhalb eines Assemblerprogramms ist flexibler geworden. Bislang mußten Prozeduren durch die Anweisung »proc« eingeleitet und durch die Anweisung »endp« beendet werden. Dieses starre Konzept war notwendig, weil über die Anweisung »proc« direkt oder indirekt der Entfernungstyp der Prozedur bestimmt wurde. Dieser legte wiederum fest, ob ein »retn« (Rückkehr von einer Prozedur mit dem Entfernungstyp »Near«) oder ein »retf« (Rückkehr von einer Prozedur mit dem Entfernungstyp »Far«) assembliert wurde. Da in der neuen MASM-Version die Befehle »retn« und »retf« direkt zur Verfügung stehen, lassen sich Prozeduren jetzt auch ohne die Proc-Anweisung definieren:

```
sub_lab1:
...
Anweisungen
...
retn
```

Eine weitere Arbeitserleichterung stellen die Umgebungsvariablen »masm« und »include« dar. Die Umgebungsvariable »include« dient dazu, den Suchpfad für Include-Dateien festzulegen. Die Umgebungsvariable »masm« speichert häufig benutzte Assembleroptionen, die bei jedem Aufruf des Assemblers automatisch gesetzt werden

```
set masm=/LA/ZD
```

Diese Anweisung bewirkt das automatische Setzen der Optionen »/LA« und »/ZD« bei jedem Aufruf des Assemblers. Hierbei handelt es sich übrigens ebenfalls um neu hinzugekommene Optionen. Die Option »/LA« (List All) bewirkt, daß der Assembler zusätzlich eine Programmlistingdatei erzeugt, in der auch die Wirkungen der vereinfachten Segmentanweisungen sichtbar werden. Wen es also interessiert, was bei den vereinfachten Segmentanweisungen im einzelnen passiert, sollte es mit dieser Option versuchen.

Die Option »/ZD« ist erforderlich, um eine Programmdatei für das Arbeiten mit dem symbolischen Debugger Codeview, der ab der Version 5.0 im Makroassemblerpaket enthalten ist, vorzubereiten. Die Option bewirkt, daß der Assembler symbolische Informationen über die Objektdatei überträgt. Dazu gehören zum einen Informationen über Größe und Typ der verwendeten Symbole und zum anderen Zeileninformationen, damit ein Debugging auf Quelltextebene möglich wird.

Erwähnenswert ist auch die Option »/W«. Mit ihr läßt sich die sogenannte Fehlerstufe einstellen. Die neue MASM-Version verfügt über wesentlich differenziertere Fehlermeldungen als die Version 4.0. Zusätzlich wurden alle möglichen Fehler in drei Kategorien eingeteilt:

- Kategorie 0: »Severe errors« (schwere Fehler)
- Kategorie 1: »Serious warnings« (widersprüchliche Angaben im Programm oder nicht einwandfreie Programmiertechniken)
- Kategorie 2: »Advisory warnings« (Anweisungen, die zu ineffektivem Code führen)

Lediglich »Severe errors« (schwerwiegende Fehler) führen dazu, daß keine Objektdatei erzeugt wird. Alle übrigen Warnungen zeigt der Assembler zwar an, sie haben aber keine weiteren Konsequenzen.

Mehr Spielraum beim Programmieren durch differenzierte Fehlermeldungen

Normalerweise steht die Fehlerstufe auf 1, das heißt, es werden nur Fehler der Kategorie 0 und 1 angezeigt. Der Vorteil dieser Option liegt in einem größeren Spielraum für den Programmierer. So lassen sich Typenfehler, wie zum Beispiel ein fehlender Ptr-Operator, unterdrücken. Fehler dieser Art führten unter der alten MASM-Version noch zu einem Abbruch des Assembliervorgangs, da es sich um schwerwiegende Fehler handelte. Bei der neuen Assemblerversion fallen sie in die Kategorie »Serious Warnings« und können durch Heraufsetzen der Fehlerstufe auf 0 völlig unterdrückt werden.

Der wohl gewichtigste Vorzug der MASM-Version 5.1, der sie auch gegenüber Konkurrenzprodukten wie dem Turbo Assembler 1.0 oder Optasm von SLR Sytems abhebt, ist die Tatsache, daß MASM 5.1 auch in einer OS/2-Version vorliegt. Daher muß der Programmierer den Assembler unter OS/2 nicht in der Kompatibilitätsbox ablaufen lassen.

Es zeigt sich, daß sich Microsoft für die Version 5.1 des MASM einiges einfallen ließ, um die Erzeugung von Assemblerprogrammen zu vereinfachen. Zwar ist auch die Version 5.1 noch keineswegs perfekt, aber sie stellt einen Schritt in die richtige Richtung dar, um die Assemblerprogrammierung auch für Hochsprachenprogrammierer attraktiv zu machen.

(Peter Monadjemi/zi)

MASM 5.0

- 80386- und 80387-Unterstützung
- vereinfachte Segmentanweisungen
- differenziertere Fehlermeldungen; Einteilung der Fehler in drei verschiedene Kategorien
- neue Umgebungsvariablen »masm« und »include«
- zusätzliche Optionen
- Prozeduren lassen sich nun auch ohne die Proc-/Endp-Anweisungen definieren, da die Befehle »retn« und »retf« zur Verfügung stehen
- Fließkommazahlen werden standardmäßig im IEEE-Format dargestellt und nicht mehr im Microsoft-internen Format
- zusätzliche Makrobibliothek; sie enthält Makros zur leichteren Einbindung von MS-DOS- und BIOS-Funktionen sowie zur Einbindung von Assemblerroutinen in Hochsprachen
- verbesserte Auswertung von Textmakros
- symbolischer Debugger Codeview

MASM 5.1

- verbessertes Hochsprachen-Interface, das heißt, die wichtigsten Anweisungen, die in der Version 5.0 noch als Makros in der Datei »mixed.inc« enthalten waren, sind nun in den Assembler integriert
- neue Textkonstanten »@Cpu«, »@Version« und »@Wordsize«
- String-Anweisungen »substr«, »catstr«, »sizestr« und »instr« für die Verarbeitung von Textkonstanten
- neue Anweisung »elseif«; durch sie wird der Aufbau von verschachtelten bedingten Anweisungen wesentlich übersichtlicher
- lokale Labels
- im Assemblerpaket ist nun der Microsoft Editor enthalten
- OS/2-Version verfügbar

Tabelle. Die wichtigsten Neuerungen des MASM in der Übersicht

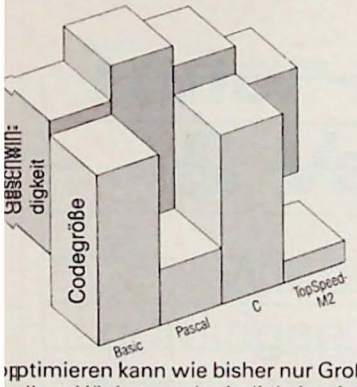
Verpassen Sie nicht den Anschluß!

Immer mehr Programmierer steigen mit TopSpeed M2 auf die neue Höchstgeschwindigkeit um, denn mit Tempo macht das Programmschreiben wieder richtig Spaß: Fehler halten kaum noch auf, neue Ideen verwirklichen Sie sofort, und Ihr fertiges Programm überzeugt durch optimale Performance.

Und wann steigen Sie in den Hochgeschwindigkeitszug TopSpeed M2 ein?

TopSpeed Modula-2

Ein Compiler erzeugt aus Ihrem Programm für den Prozessor Maschinencode, der möglichst klein sein soll: Ihr Programm wird schneller geladen, kann mehr Daten im Hauptspeicher erfassen und läuft schneller ab.



Ein "intelligenter" Compiler optimiert daher den erzeugten Code, das heißt, er faßt doppelte Programmteile zusammen, vermeidet Sprünge zwischen Speichersegmenten und verwendet die effektivsten Maschinenbefehle zur Ausführung.

Der TopSpeed-M2-Compiler hat in punkto Größe und Ablaufgeschwindigkeit des erzeugten Maschinencodes derzeit die Nase vorn, da er auch auf dem PC so gut und intelligent optimieren kann wie bisher nur Großrechnersysteme. Überzeugen Sie sich selbst: Höchstgeschwindigkeit mit TopSpeed-M2.

Nur bei uns

- Offizielle deutsche Compilerversion
- Aktuelle, neueste Versionsnummer
- Update-Service zum Selbstkostenpreis
- TopSpeed-Kundenzeitschrift
- Telefon-Helpline
- Super-Paketpreis für Compiler und Debugger
- B-Tree-ISAM & Display-Manager, Dateiverwaltlung und Maskengenerator im Lieferumfang

LAUER & WALLWITZ Software par excellence! L&W

DMV Daten- und Medienverlag

Zusatzprodukte

TechKit

Enthält Routinen zur maschinennahen Programmierung und einen in die Compileroberfläche integrierbaren Assembler:

Unter anderem Tools zur Programmierung speicherresidenter Programme (TSR), Routinen zur Ansteuerung der seriellen Schnittstelle, einen Overlay-Manager, Funktionen zum EMS-Handling, BIOS-Interface, Locator für PROM-Programme sowie weitere Modula-2-spezifische Utilities.

VID - Visual Interface Debugger

Komfortable Fehlersuche mit dem Source-Level-Debugger zum TopSpeed M2: Einzelschrittmodus, Trace-Möglichkeiten, Zugriff auf alle Variablen und auch Strukturen, Zeiger und Aufzählungstypen, in symbolischer Form. Besonders einfach ist auch der Low-Level-Zugriff auf Register und Speicherbereiche.

Um Programme kontrolliert ablaufen zu lassen, können verschiedene Arten von Unterbrechungspunkten im Quellcode gesetzt und sowohl Variablen als auch Modula-2-Ausdrücke während des Programmlaufs beobachtet werden. Protokollierung des Zugriffs auf Variablen und eine Stapelanzeige der Prozedur-Aufrufsverschachtelung sind gleichfalls möglich.

Der Debugger beinhaltet einen symbolischen Disassembler und einen "Profiler", der eine Analyse des Programm-Laufzeitverhaltens erlaubt.

Aus dem DMV-Verlag

MaskEdit Plus 5.0
Professioneller, schneller und komfortabler Maskengenerator Version für Topspeed Modula-2 **DM 99, – ***

GEMInterface PC
Leistungsfähiges Toolkit zur Erstellung eigener GEM-Applikationen Version für Topspeed Modula-2 **DM 198, – ***

Minimale Systemanforderungen
PC/XT/AT mit 384 kByte RAM, zwei Diskettenlaufwerke oder Festplatte, PC-/MS-DOS ab Version 2.0

TopSpeed Compiler + ISAM & Display Manager	**DM 348, – ***
TopSpeed TechKit	**DM 248, – ***
TopSpeed Debugger	**DM 248, – ***
Paketpreis für Compiler und Debugger	**DM 555, – ***
TopSpeed Compiler, OS/2-Version	**DM 698, – ***

* Unabhängig von der Anzahl der bestellten Programme berechnen wir für das Inland 4,- DM bzw. für das Ausland 6,- DM Porto und Verpackung. – Bitte benutzen Sie die Bestellkarte –

DMV-Verlag · Postfach 250 · 3440 Eschwege

Arithmetikprozessoren – Heinzelmännchen der CPU

Was für den Schüler der Taschenrechner, ist für die Zentraleinheit (CPU) der Arithmetikprozessor. Wer viel mit Grafik oder mathematischen Berechnungen konfrontiert wird, weiß den Wert eines Arithmetikprozessors mit Sicherheit zu schätzen. Die Geschwindigkeit, mit der ein 80x87 eine komplizierte mathematische Aufgabe löst, übersteigt die Fähigkeiten eines 80x86-Prozessors bei weitem.

Ein Arithmetikprozessor vom Typ 80x87 besitzt eine völlig andere Struktur als die Zentraleinheit (Prozessor) Ihres Computers. Seine acht gleichartigen 80 Bit breiten Datenregister, »st« bis »st(7)«, haben keine andere Aufgabe, als die geladenen Werte mathematisch miteinander zu verbinden oder zu vergleichen. Die große Bitbreite gewährleistet zudem einen hohen Genauigkeitsfaktor des Resultats. Während der Prozessor (80x86) nur einen Teil seiner Leistung dem geladenen Programm widmen kann, konzentriert sich ein Arithmetikprozessor (80x87) einzig auf die vom Anwender gestellte Aufgabe.

Vergleicht man die vom Prozessor beziehungsweise Arithmetikprozessor aufzuwendenden Systemtakte für die vier Grundrechenarten Addition, Subtraktion, Multiplikation und Division, so ergeben sich zeitliche Vorteile für den Prozessor. Allerdings nur, wenn es sich nicht um Fließkommazahlen mit einer Länge von 32 oder 64 Bit handelt. Und das führt direkt zum eigentlichen Zweck eines Arithmetikprozessors: die Arbeit mit Fließ- oder Gleitkommazahlen.

Was unterscheidet eine Festkommazahl von einer Gleitkommazahl? Eine Festkommazahl ist eine ganze, ungebrochene Zahl, bei der hinter der letzten Stelle ein Komma angenommen wird. Zum Beispiel ist » + 11235« eine Festkommazahl. Alle Ziffern stehen vor dem »festen« Komma. Läßt man das Komma zwischen die Ziffern »gleiten«, wird beispielsweise die Gleitkommazahl » + 112,35« daraus. Mit dieser zweigeteilten Zahl läßt sich aber kein einzelnes Register eines 80x86-Prozessors laden – wohl aber eines der Datenregister des 80x87-Arithmetikprozessors. Das Register übernimmt die Zahl allerdings nicht in dieser Form, sondern sie wird vorher in die »temporary real number« (» + 0,11235x10 $^{+3}$«) konvertiert. Es erfolgt also eine Zerlegung der Zahl in zwei Einheiten, die Mantisse und den Exponenten, der auch als Charakteristik bezeichnet wird. Im Register würde die Zahl dann so aussehen: »11235 + 3« (in binärer Form, versteht sich). Alle Zahlen sind vorzeichenbehaftet. Der Exponent gibt an, um wie viele Stellen das Komma nach links (–) oder rechts (+) zu verschieben ist.

Das »Temporary Real Format«, in dem der Coprozessor alle Werte verarbeitet, ist ein 10 Byte breites IEEE-Format (Institute of Electrical and Electronic Engineers, vergleichbar mit unserem VDE). In diesem Format ist das Vorzeichenbit – 0 für positiv oder 1 für negativ – immer das höchste Bit, dann folgen Exponent und Mantisse, unabhängig davon, aus wie vielen Bytes die Zahl besteht.

Die IEEE-gerechte Darstellung gestattet die einfache Bearbeitung von sehr kleinen und auch sehr großen Zahlen, wie sie häufig im mathematisch-wissenschaftlichen Bereich vorkommen. Denken Sie nur an Logarithmen. Im Datenregister belegt die Mantisse die Bit 0 bis 3Fhex und der Exponent die Bit 40hex bis 4Fhex, wobei das höchste Bit (4Fhex) das Vorzeichen beinhaltet. Damit stehen also der Mantisse 64 Bit und dem Exponenten 15 Bit zur Verfügung. Das bedeutet, daß der Arithmetikprozessor in einem Schritt eine positive oder negative Zahl von einer Trillion minus 1 im BCD-Format (»Binary Coded Decimal«) verarbeiten kann – eine 18stellige Zahl mit Vorzeichen.

Der Aufbau der acht Datenregister des Arithmetikprozessors gleicht einem Stapel (Bild 1) und wird auch »Data Register Stack« genannt. Der Stackpointer zeigt normalerweise auf

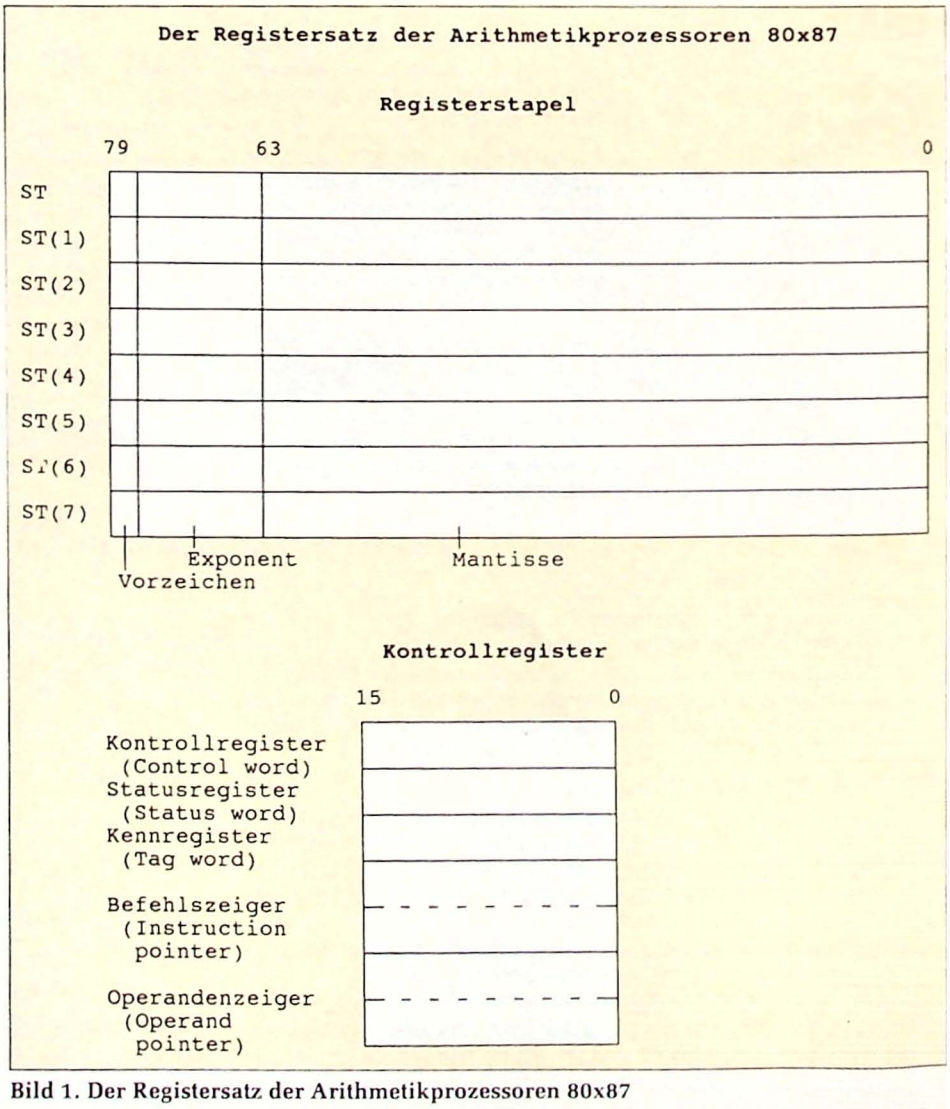


Bild 1. Der Registersatz der Arithmetikprozessoren 80x87

das oberste Register »st(0)« (oder einfach als »st« bezeichnet). Darunter liegen die Register »st(1)« bis »st(7)«. Außer den Datenregistern besitzt ein Arithmetikprozessor der 80x87-Familie noch sieben Kontrollregister von je 16 Bit Breite. In ihnen werden das »Control Word«, »Status Word«, »Tag Word«, der »Instruction Pointer« und »Operand Pointer« abgelegt. Die beiden Zeiger benutzen je 16 Bit für die Segment- und Offset-Adresse. Die Kontroll- und Statusworte (1 Byte = 8 Bit, 1 Wort = 16 Bit) sind bitweise gegliedert. Ein oder zwei Bit enthalten immer eine bestimmte Information, wie es auch im Flag-Register der Zentraleinheit der Fall ist (Bild 2).

Wenn Sie in einer Hochsprache programmieren, übersetzt der Pascal- oder C-Compiler auf Anweisung die mathematischen Quellcodebefehle in die »Mnemonics« des 80x87, oder er »emuliert« sie. Das heißt, sofern Ihr Rechner keinen Arithmetikprozessor besitzt, werden diese Befehle so gestaltet, daß der 80x86-Prozessor sie ausführen kann. Nur benötigt der Prozessor dazu bis zu zehnmal mehr Zeit für die Ausführung.

Anders in Assembler. Hier gibt es keine Bibliothek »emu.lib«, in der ein Linker »emulierte« Befehle vorfindet. Entweder nimmt man die »Übersetzung« der Coprozessorbefehle selbst vor oder man spricht ihn direkt an. Dazu muß er natürlich vorhanden sein. Der Befehlssatz eines 80x87 besteht aus zirka 100 »Mnemonics«, von denen einige dem 80387 vorbehalten sind, während andere nur für den 8087 gelten. Alle diese »Mnemonics« beginnen mit »F« als Unterscheidungskennzeichen für den Assembler (MASM, TASM etc.), daß es sich dabei um Coprozessorbefehle handelt. Unter ihnen befinden sich Anweisungen, wie zum Beispiel »fyl2xp1«, die schon eine kleine Formel enthalten (»Z = Y $\log_2(X + 1)$«). In diesem Beispiel wird das Register »st« mit »X«, »st(1)« mit »Y« geladen und das Ergebnis »Z« wiederum »st« entnommen.

Einschließlich der Ladeoperationen aus und in den Speicher benötigt ein mit 10 MHz getakteter 8087 etwa eine zehntausendstel Sekunde zur Lösung dieser Aufgabe. Ein mit gleicher Frequenz arbeitender 80387 ist noch zirka 2 1/2mal schneller. Wenn Sie nun noch berücksichtigen, daß die Operanden »X« und »Y« bis zu 64 Bit lang sein dürfen, bekommen Sie eine ungefähre Vorstellung von der Leistungsfähigkeit eines Arithmetikprozessors. Die normale Bildschirmausgabe der Operanden und des Resultats durch den Prozessor nimmt da bereits mehr Zeit in Anspruch.

Die beiden Typen 8087 und 80287 unterscheiden sich nur sehr geringfügig in der Ausführungsgeschwindigkeit der Befehle. Beide benötigen die gleiche Anzahl von Takten, um eine Anweisung auszuführen. Der innere Aufbau des 80387 hingegen ist völlig anders gestaltet als der seiner Vorgänger. Für einige Befehle benötigt er nur ein Fünftel der Takte des 8087 beziehungsweise 80287, für andere wiederum mehr als die doppelte Anzahl.

Wie schnell ein Arithmetikprozessor in Realzeit arbeitet, ist selbstverständlich abhängig von der Geschwindigkeit des Rechners, in dem er eingesetzt ist. Ein 80287 in einem mit 20 MHz getakteten AT ist mehr als fünfmal so schnell wie ein 8087 in einem PC mit 4,77 MHz Taktfrequenz. Sind beide in 10-MHz-Rechnern installiert, ist der 80287 im AT allerdings nicht schneller als der 8087 im PC.

Obwohl Prozessor und Arithmetikprozessor ihre Aufgaben unabhängig

Toner-Service & Vertieb

Wir bieten zu günstigen Konditionen Telefax- und Kopiergeräte, Drucker + Zubehör,

z.B. Toner, Schreibmaschinen- u. Druckerbänder, Laserkartuschen und Papier

Tel. 0 30/3 32 00 21
Fax 0 30/3 32 00 24

DIE KAUFMÄNNISCHE KOMPLETTLÖSUNG FÜR HANDWERK UND HANDEL (auch Bauhaupt u. Nebengewerbe)

Die Programme sind:
- Bundesweit ca 20.000 mal installiert
- vom Finanzamt und Krankenkassen geprüft und abgenommen
- Alle Teilprogramme arbeiten zusammen und brauchen lediglich einen Stammdatensatz
- Von einem Hauptmenü (Fenstertechnik) alle Programme aufrufbar
- Und das Wichtigste, egal wo Sie sich im Programm befinden, das Handbuch erscheint stets auf Tastendruck am Monitor
- Mandantenfähig (bis zu 10 Unternehmen gleichzeitige Bearbeitung)

Wir schicken Ihnen gerne Prospektmaterial und informieren Sie völlig unverbindlich.

Einarbeitung, Schulung und Hardwarelieferung auf Wunsch – ebenfalls preisgünstig –

UNTERNEHMENSBERATUNG BARTSCH & KLOFT
5439 Guckheim, Kirchstr. 9
0 64 35/32 10

DIE KAUFMÄNNISCHE KOMPLETTLÖSUNG FÜR HANDWERK UND HANDEL (auch Bauhaupt u. Nebengewerbe)

B & K Auftragsbearbeitung
Angebote, Kostenvoranschläge, Werkstattaufträge, Lieferscheine, automatische Übernahme in Fibu usw. usw 395,–

B & K Finanzbuchhaltung
Nach neuem Bilanzrichtliniengesetz, Mit Datev-Kontenrahmen SKR 04 usw. usw 395,–

B & K Lohnbuchhaltung
komplette Auswertungen incl. ZVK, Krankenkassenlisten, Wintergeld, Bauurlaubsberechnung, Überweisungsträger usw. usw 395,–

B & K Textverarbeitung
mit Serienbrieffunktion, Kopf- und Fußzeilenverwaltung usw. usw 195,–

ALS KOMPLETTPAKET NUR DM 995,–

UNTERNEHMENSBERATUNG BARTSCH & KLOFT
5439 Guckheim, Kirchstr. 9
0 64 35/32 10

voneinander bewältigen, müssen sie synchron arbeiten. Das heißt, die Zentraleinheit oder der Coprozessor warten häufig, bis der jeweils andere Teil seine momentane Aufgabe gelöst hat und ein eventuelles Resultat zur Verfügung steht. Diese Synchronisation ist äußerst wichtig für den Programmablauf. Manche Assembler (beispielsweise MASM) berücksichtigen diesen Umstand schon beim Assemblieren und fügen selbständig einen Wait-Befehl an den entsprechenden Stellen ein. Dieser weist den Arithmetikprozessor an, zu warten, bis der Prozessor seine Aufgabe beendet hat.

Für den Wait-Befehl an die Zentraleinheit muß der Programmierer selbst sorgen. Über Bit 15 im Statuswort des Kontrollregisters des 80x87 erhält er dazu Informationen. Ist dieses Bit gesetzt, arbeitet der Arithmetikprozessor noch an einem Befehl. Da die Kontrollregister aber fast nur für Systemprogrammierer von Interesse sind, erübrigt es sich, in diesem Rahmen weiter darauf einzugehen.

Wer häufig hochauflösende Grafikprogramme benutzt, kennt das Problem des langsamen Bildaufbaus detaillierter Grafiken. Auch hier lohnt sich der Einsatz eines Arithmetikprozessors, denn grafische Darstellungen beruhen zum größten Teil auf Positionsberechnungen von Bildpunkten und Linien in Relation zueinander.

Wie die Zentraleinheit wird auch der Arithmetikprozessor in Assembler mit »Mnemonics« programmiert

Ein 80x87 berechnet die Lage der Bildpunkte eines Kreises oder eines Bogens bedeutend schneller als ein 80x86. Der Faktor »Pi«, den man bekanntlich zur Kreisberechnung benötigt, ist für den Arithmetikprozessor auch ohne vorherige Definition ein konstanter Wert.

Der Unterschied zwischen den mathematischen Coprozessoren 8087 und 80287 besteht hauptsächlich darin, daß der 80287 auch im »Protected Mode« läuft. Um ihn in diesen Modus zu schalten, benutzt man den nur für ihn relevanten Befehl »fsetpm«. Die beiden Zeiger in den Kontrollregistern verweisen dann – wie auch beim 80387 – auf Segmentselektoren und nicht, wie beim 8087, direkt auf das Segment einer Adresse. Trifft der 80387 auf den Befehl »fsetpm«, interpretiert er ihn als »fnop«, das heißt, er führt keine Operation aus.

Die Interrupts betreffenden Befehle, wie zum Beispiel »fdisi« und »feni«, gelten nur für den 8087, da weder der 80287 noch der 80387 über eine »Interrupt Enable Mask« verfügen. Befinden sich derartige Befehle im Quellcode, werden sie von diesen beiden Typen ignoriert. Der Befehlssatz des 80387 wurde um sieben Befehle erweitert. Sie betreffen mathematische Vergleiche und die trigonometrischen Funktionen »Sinus« und »Cosinus«. Außerdem unterscheidet der 80387 nicht zwischen »Protected Mode« und »Real Mode«. Bei der Adressierung wird immer davon ausgegangen, daß sich das System im »Protected Mode« befindet.

Die Zusammenarbeit von Prozessor und Arithmetikprozessor unterliegt bestimmten Regeln. Der mathematische Coprozessor hat keinen Zugriff auf die direkte Ein- und Ausgabe. Ihm steht lediglich der Speicher zur Verfügung. Der Ablauf einer Operation erfolgt gewöhnlich in drei Schritten:

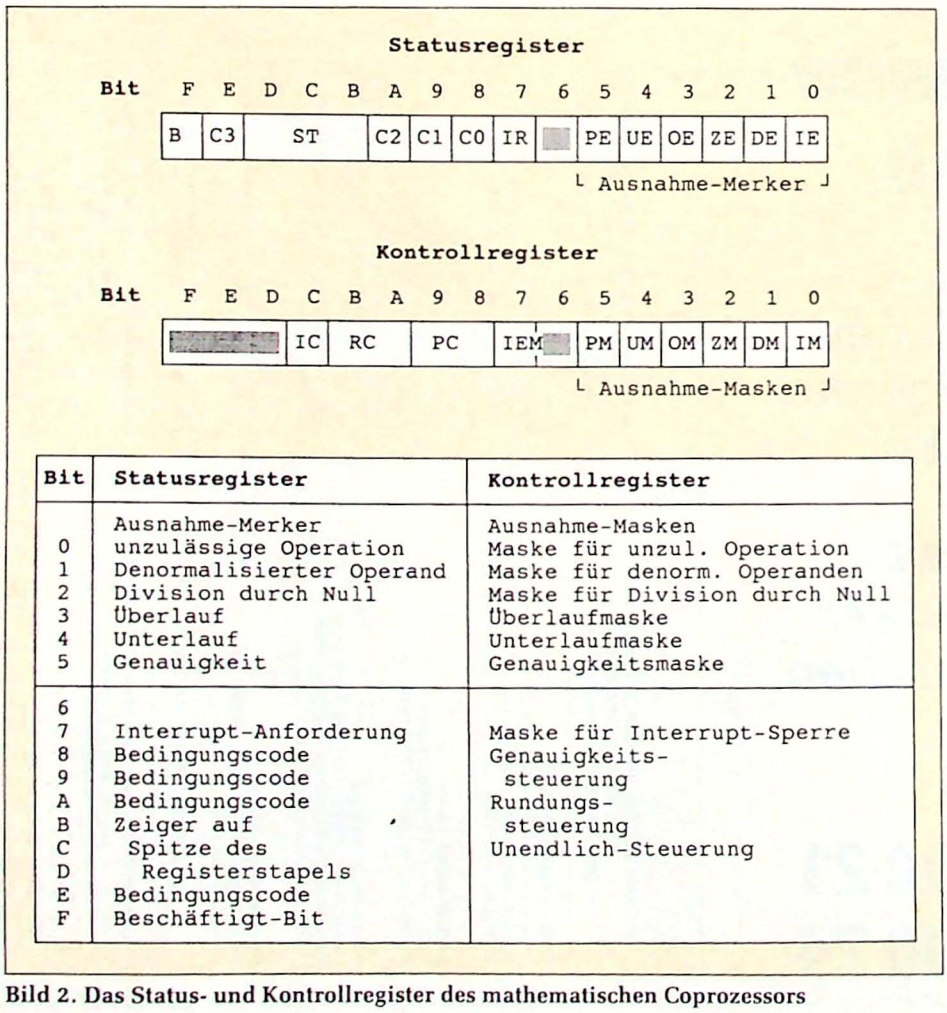
- Daten aus dem Speicher in die Datenregister laden
- Daten verarbeiten
- Daten/Resultat aus dem(n) Register(n) in den Speicher laden

Wenn der Arithmetikprozessor die Schritte eins und drei ausführt, muß sichergestellt sein, daß die Zentraleinheit nicht zu genau derselben Zeit auf die von ihm benutzten Speicheradressen zugreift. Während der Datenverarbeitung (Schritt 2) unterliegt der Prozessor keinen Einschränkungen. Er kann unterdessen Interrupts bearbeiten oder sonstige Aufgaben erledigen.

Mit einem Wait-Befehl hat der Programmierer dafür Sorge zu tragen, daß der Prozessor erst dann auf das Resultat des Arithmetikprozessors zugreift, wenn es im Speicher steht. Andernfalls übernimmt er den Wert, der vor dem Resultat an dieser Adresse abgelegt war.

Ein 80x87-Arithmetikprozessor kennt außer einigen logarithmischen Konstanten nur noch »Pi«, 1 und 0. Wollen Sie im Quellcode definierte Konstanten verarbeiten, müssen Sie Speicher zuweisen, indem Sie eine Variable benennen. Diese Variable initialisieren Sie mit dem Wert der Konstanten. Dann laden Sie den Wert aus dem Speicherbereich, der der Variablen zugewiesen wurde, in ein Datenregister des Arithmetikprozessors. Da Konstanten keine festen Speicheradressen besitzen, kann der Arithmetikprozessor nicht direkt auf sie zugreifen.

(Jürgen Hochstädt/zi)



Bit	Statusregister	Kontrollregister
	Ausnahme-Merker	Ausnahme-Masken
0	unzulässige Operation	Maske für unzul. Operation
1	Denormalisierter Operand	Maske für denorm. Operanden
2	Division durch Null	Maske für Division durch Null
3	Überlauf	Überlaufmaske
4	Unterlauf	Unterlaufmaske
5	Genauigkeit	Genauigkeitsmaske
6		
7	Interrupt-Anforderung	Maske für Interrupt-Sperre
8	Bedingungscode	Genauigkeits-
9	Bedingungscode	steuerung
A	Bedingungscode	Rundungs-
B	Zeiger auf	steuerung
C	Spitze des	Unendlich-Steuerung
D	Registerstapels	
E	Bedingungscode	
F	Beschäftigt-Bit	

Bild 2. Das Status- und Kontrollregister des mathematischen Coprozessors

Die (un-)heimlichen Helfer von MS-DOS

MS-DOS gehört zu den zuverlässigsten und vielseitigsten Betriebssystemen, die es gibt. Einen Großteil seines Erfolgs hat das Betriebssystem dem Umstand zu verdanken, daß es leicht erweiterbar ist. Die Basis für diese Struktur bilden die sogenannten Gerätetreiber, die sämtliche Ein- und Ausgaben steuern.

Was sind eigentlich Gerätetreiber, und welche Aufgaben übernehmen sie in einem PC ? Um diese Frage zufriedenstellend zu beantworten, müssen zuerst grundlegende Sachverhalte erörtert werden.

Um mit Computern sinnvoll zu arbeiten, brauchen Sie ein Betriebssystem. Ein Betriebssystem ist eine Sammlung von Routinen und Unterprogrammen, die ein Anwendungsprogramm in die Lage versetzen, alle Fähigkeiten und Bestandteile eines Computers auszunutzen. Damit das Anwendungsprogramm auf vielen Computern lauffähig ist, stellt das Betriebssystem eine standardisierte Schnittstelle zur Verfügung. Das hat mehrere Vorteile: Erstens müssen die Programmierer nicht die Hardware des Computers direkt ansprechen, sondern sie arbeiten mit den standardisierten Prozeduren. Das garantiert, daß die Programme auch auf späteren Versionen des Betriebssystems lauffähig sind, da die Entwickler die Benutzerschnittstelle nicht ändern, sondern höchstens um zusätzliche Funktionen erweitern.

Zweitens ist jedes Anwendungsprogramm auf verschiedenen Computern mit kompatiblen Mikroprozessoren lauffähig, obwohl sich die Betriebssystemversionen stark voneinander unterscheiden. Die installierte Software ist also hardwareunabhängig. Da ein Betriebssystem sehr pauschal gehalten ist, deckt es natürlich auch nur die Ansteuerung der Standard-Ein- und Ausgabegeräte ab. An diesem Punkt beginnt die Aufgabe der Gerätetreiber.

Ein Gerätetreiber (englisch »Device Driver«) ist ein hardwareabhängiges Programm, das ein Peripheriegerät steuert, die Kontrolle über diese zusätzliche Einheit an die standardisierte Schnittstelle des Betriebssystems anpaßt und somit ein Teil des Betriebssystems wird. Da der Gerätetreiber hardwareabhängig ist, läßt er sich natürlich nicht so leicht auf andere Systeme übertragen.

Gerätetreiber sind die Verbindung des Betriebssystems zur Außenwelt

Ein Gerätetreiber ist also im Grunde nichts anderes als eine standardisierte Methode der Datenübergabe. Mit Gerätetreibern realisieren Sie bei entsprechender Programmierung viele Anwendungen. So lassen sich beispielsweise RAM-Disks einrichten, ROM-Laufwerke ansprechen, Meßgeräte ansteuern oder komplette Standardeinheiten wie etwa die Tastatur neu belegen.

Das Betriebssystem stellt – wie schon gesagt – mehrere Standardtreiber zur Verfügung, deren Namen und Funktionen in Tabelle 1 aufgeführt sind. MS-DOS ist sinnvollerweise so aufgebaut, daß jeder Benutzer die Standardtreiber durch eigene Routinen ersetzen kann.

Es wird grundsätzlich zwischen zwei Klassen von Gerätetreibern unterschieden: den zeichenorientierten und den blockorientierten. Die zeichenorientierten Treiber steuern Geräte wie beispielsweise den Monitor, die Tastatur oder den Drucker. Die blockorientierten Treiber kontrollieren hingegen die Peripherie wie Diskettenlaufwerke, Festplatten oder CD-ROMs.

Gerätetreiber bieten die vielseitigsten Anwendungsmöglichkeiten

Als Beispiel für einen installierbaren Treiber ist auf der MS-DOS-Diskette der ANSI-Treiber »ansi.sys« gespeichert, der die Tastatureingabe und Bildschirmausgabe wesentlich erleichtert. Sie binden ihn, wie auch jeden anderen Treiber, mit der Anweisung

```
device = (Name des Treibers)
```

also in diesem Fall

```
device = ansi.sys
```

in die MS-DOS-Umgebung ein, die zuerst in den Arbeitsspeicher geladen wird. Im Gegensatz zu dem sonst üblichen Verfahren leitet das Betriebssystem nun alle Ausgaben über den ANSI-Treiber um.

Dieser Treiber teilt die umgeleiteten Zeichen in zwei Gruppen auf; die

Reservierte Gerätenamen

Name	Funktion
NUL	Null-Device
?	In »config.sys« vom Anwender definierte Treiber
CON	Konsole (Tastatur und Monitor)
AUX	Serielles Zusatzgerät (Com1:)
PRN	Paralleles Zusatzgerät (Lpt1:)
[Uhr]	Zeit-Device (Systemuhr mit 18,2 Impulsen pro Sekunde)

Tabelle 1. Einige Gerätetreiber sind vom Betriebssystem vorgegeben

Gerätetreiber-Handler

Adresse	Länge	Name	Funktion
0hex	32 Byte	Nächster Gerätetreiber	Adresse des nächsten Gerätetreibers beziehungsweise 0FFFFFFFFhex für den letzten Treiber
4hex	16 Byte	Geräte-Attribut-Code	siehe Tabellen 3 und 4
6hex	16 Byte	Geräte-Strategie-Zeiger	Offset-Adresse, die auf Strategieroutine zeigt
8hex	16 Byte	Geräte-Interrupt-Zeiger	Offset-Adresse, die auf Interrupt-Routine zeigt
Ahex	8 Byte	Gerätename	Gerätename in ASCII-Form; bei Blocktreibern bestimmt das erste Byte die Anzahl der Geräteeinheiten

Tabelle 2. Der Gerätetreiber-Handle ist für die Initialisierung notwendig

Geräte-Attributcodes für zeichenorientierte Treiber

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Name	Bedeutung
x	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	Eingabe (Konsole)	Gelöscht: Treiber ist nicht Standard-Eingabegerät. Gesetzt: Treiber ist Standard-Eingabegerät.
–	x	–	–	–	–	–	–	–	–	–	–	–	–	–	–	Ausgabe (Konsole)	Gelöscht: Treiber ist nicht Standard-Ausgabegerät. Gesetzt: Treiber ist Standard-Ausgabegerät.
–	–	x	–	–	–	–	–	–	–	–	–	–	–	–	–	Null-Device	Gelöscht: Treiber ist kein Null-Device. Gesetzt: Treiber ist Null-Device.
–	–	–	x	–	–	–	–	–	–	–	–	–	–	–	–	Uhr-Device	Gelöscht: Treiber ist kein Uhr-Device. Gesetzt: Treiber ist Uhr-Device.
–	–	–	–	x	x	x	x	x	x	x	x	x	–	–	–	Reserviert	
–	–	–	–	–	–	–	–	–	–	–	–	–	x	–	–	Ausgabe bis beschäftigt	Gelöscht: Treiber unterstützt »until busy« nicht. Gesetzt: Treiber unterstützt »until busy«.
–	–	–	–	–	–	–	–	–	–	–	–	–	–	x	–	IOCTL	Gelöscht: IOCTL wird unterstützt. Gesetzt: IOCTL wird nicht unterstützt.
–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	x	Gesetzt: zeichenorientierter Treiber	

Tabelle 3. Der Attributcode übernimmt die Klassifizierung des Geräts

eine Gruppe besteht aus den Zeichen, die auf dem Bildschirm erscheinen sollen, und die andere Gruppe ist aus Befehlen aufgebaut. Der ANSI-Treiber filtert aus den Bildschirmdaten die entsprechenden Informationen heraus und gibt sie dann an die Ausgabeeinheit weiter.

So kennzeichnet beispielsweise das vorangestellte ASCII-Zeichen mit dem Wert 27 beim ANSI-Treiber einen Befehl. Die nachfolgenden Angaben bestimmen, ob es sich beispielsweise um ein Kommando zum Löschen des Bildschirms, zum Ändern der Farbe oder zum Positionieren des Cursors handelt.

Der wohl bekannteste aller Gerätetreiber ist der Konsolentreiber »ansi.sys«

Die Verwendung des ANSI-Treibers bringt zwei wesentliche Vorteile mit sich. Der erste Vorteil liegt darin, daß anstelle der BIOS-Routinen die BIOS-ähnlichen Routinen des Treibers benutzt werden können. Des weiteren laufen alle Programme, die den ANSI-Treiber anstatt der BIOS-Routinen benutzen, auf allen MS-DOS-Computern.

Die wesentlichen Nachteile des ANSI-Treibers und aller anderen zusätzlichen Gerätetreiber bestehen einerseits in der Inkompatibilität der Programme zu den Computern, auf denen der Treiber nicht installiert ist, und andererseits in der herabgesetzten Geschwindigkeit.

Das Programmieren eines Gerätetreibers ist ziemlich aufwendig, weil Sie einen Fehler in der Struktur kaum lokalisieren können, denn das Betriebssystem installiert die Gerätetreiber automatisch, wenn »command.com« geladen wird. Es ist daher kaum möglich, diesen Vorgang mit einem Debugger oder einem ähnlichen Hilfsprogramm nachzuvollziehen. Aus diesem Grund müssen Sie beim Aufbau eines Gerätetreibers besondere Sorgfalt walten lassen. Es ist sinnvoll, den Standardaufbau eines Treibers zu übernehmen und daraufhin die Details Schritt für Schritt hinzuzufügen.

Jeder Gerätetreiber besteht grundsätzlich aus drei Teilen. Der erste Teil ist der »Device Header«, der den Namen des Treibers bestimmt und die Verbindung zu den anderen Teilen des Treibers herstellt. Der zweite Abschnitt ist die »Device Strategy«, die einen vom Betriebssystem übergebenen Datenblock namens »Request Header« kontrolliert. Als dritter und letzter Bestandteil fungiert der »Device Interrupt Handler«, der die Programmroutinen des Treibers enthält.

Die Gerätetreiber werden beim Boot-Vorgang zuerst geladen

Um den Aufbau eines Gerätetreibers zu verstehen, ist es am einfachsten, einen bereits bestehenden zu analysieren. Zu diesem Zweck finden Sie im Listing einen einfachen Geräte

Geräte-Attributcodes für blockorientierte Treiber

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Inhalt	Bedeutung
x	x	x	x	x	x	x	x	x	x	x	x	x	–	–	–	Reserviert	Gelöscht: FAT-ID-Byte nicht benutzt. Gesetzt: FAT-ID-Byte benutzt.
–	–	–	–	–	–	–	–	–	–	–	–	–	x	–	–	Begrenzer des Medium-Typs	
–	–	–	–	–	–	–	–	–	–	–	–	–	–	x	–	Kontroll-Zeichenketten bearbeiten	Gelöscht: Zeichenketten nicht erlaubt. Gesetzt: Kontroll-Zeichenketten können bearbeitet werden.
–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	Gerätetyp	Gelöscht: blockorientierter Treiber

Tabelle 4. Das Attributcode eines blockorientierten Treibers unterscheidet sich deutlich von einem Zeichentreiber

Statusfeld des Request-Headers

Bit-Nummern 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Inhalt	Bedeutung
x	x	x	x	x	x	x	x	–	–	–	–	–	–	–	–	Fehlercode	Siehe Tabelle 6
–	–	–	–	–	–	–	–	x	–	–	–	–	–	–	–	Vollbracht	Gelöscht: Operation dauert an. Gesetzt: Operation beendet.
–	–	–	–	–	–	–	–	–	x	–	–	–	–	–	–	Beschäftigt	Gelöscht: nicht beschäftigt. Gesetzt: beschäftigt (busy).
–	–	–	–	–	–	–	–	–	–	x	x	x	x	x	–	Reserviert	
–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	x	Fehler-Flag	Gelöscht: kein Fehler. Gesetzt: Fehler

Tabelle 5. Der Request Header kontrolliert die Ausführung eines Device-Kommandos

treiber, der nichts anderes ausführt, als bei seiner Initialisierung eine Meldung auf den Bildschirm auszugeben.
Die Startadresse eines Gerätetreibers ist grundsätzlich auf Null zu beziehen. Das erreichen Sie mit Hilfe des Org-Pseudobefehls (im Listing »org 0h«). Zusätzlich sind die Randbedingungen für Com-Dateien zu erfüllen, das bedeutet beispielsweise, daß Sie kein Stapelsegment definieren dürfen.
Nach der Eingabe des Quelltextes im Listing übersetzen Sie das Programm mit einem Assembler und linken es. Falls bei beiden Programmen kein Fehler aufgetreten ist, wandeln Sie das Exe-Programm mit dem Exe2bin-Utility in eine Com-Datei um. Zuletzt tragen Sie in die Datei »config.sys« den Pfad und den Namen des Gerätetreibers ein. In unserem Fall ist das die Anweisung

```
device = c:\beispiel.com
```

wenn sich das Betriebssystem im Hauptverzeichnis (»Root«) auf der Festplatte befindet. Danach starten Sie Ihren Computer neu. Bei fehlerfreiem Betrieb erscheint die Einschaltmeldung des Beispieltreibers nach kurzer Zeit auf dem Bildschirm, und das Betriebssystem befindet sich in der Kommandoebene.

Ein Gerätertreiber muß immer in einer Com-Datei-kompatiblen Form vorliegen

Als ersten Teil des Gerätetreibers finden Sie im Listing den sogenannten »Device Header«, dessen Aufbau Tabelle 2 wiedergibt. Der Header ist 18 Byte lang und besteht aus fünf Feldern. Die ersten beiden Felder geben die Position des folgenden Gerätetreibers an. Wenn Sie in beiden Feldern vor der Initialisierung den Wert -1 festlegen, übernimmt das Betriebssystem die Adressierung automatisch. Nach der Initialisierung legt der Wert -1 fest, daß es sich bei dem aktuellen Treiber um den zuletzt geladenen handelt.

Feld zwei enthält das für den jeweiligen Treiber verantwortliche Attributbyte, dessen Inhalt für zeichenorientierte Treiber in Tabelle 3 und für blockorientierte Treiber in Tabelle 4 aufgelistet ist. Das Attributbyte ist für die Klassifizierung des Treibers verantwortlich.

Das dritte und vierte Feld bestimmen die Adressen der Strategie- und der Interrupt-Routine.

Das fünfte und letzte Feld enthält schließlich den Namen des Gerätetreibers. Dieser muß genau acht Zeichen lang sein. Fehlende Buchstaben füllen Sie mit Leerzeichen auf. Bei einem blockorientierten Treiber ist nur das erste Byte signifikant, das die Anzahl der Einheiten enthält, die der Treiber bearbeiten kann.

Die Strategieroutine wird nur bei der Initialisierung aufgerufen

Der zweite Teil des Gerätetreibers ist der Strategiecode, der aus fünf Programmzeilen aufgebaut ist. Sie finden ihn im Listing unter dem Namen »strategie«. Wenn das Betriebssystem einen Treiber lädt, baut es einen Datenblock namens »Request Header« auf, der zwei Funktionen erfüllt. Zum einen agiert er als Datenbereich für MS-DOS und zum anderen stellt er die Schnittstelle dar, durch welche das aufrufende Programm Informationen mit dem Gerätetreiber austauscht.

Bit-Nummern 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Fehlername
–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	x	0: schreibgeschützt
x	–	–	–	–	–	–	–	–	–	–	–	–	–	–	x	1: unbekannte Geräteeinheit (Unit)
–	x	–	–	–	–	–	–	–	–	–	–	–	–	–	x	2: Laufwerk nicht bereit
x	x	–	–	–	–	–	–	–	–	–	–	–	–	–	x	3: unbekanntes Kommando
–	–	x	–	–	–	–	–	–	–	–	–	–	–	–	x	4: CRC-Fehler
x	–	x	–	–	–	–	–	–	–	–	–	–	–	–	x	5: falsche Länge der Request-Struktur (Laufwerk)
–	x	x	–	–	–	–	–	–	–	–	–	–	–	–	x	6: Fehler beim Suchen
x	x	x	–	–	–	–	–	–	–	–	–	–	–	–	x	7: unbekanntes Medium
–	–	–	x	–	–	–	–	–	–	–	–	–	–	–	x	8: Sektor nicht gefunden
x	–	–	x	–	–	–	–	–	–	–	–	–	–	–	x	9: Drucker ohne Papier
–	x	–	x	–	–	–	–	–	–	–	–	–	–	–	x	A: Schreibfehler
x	x	–	x	–	–	–	–	–	–	–	–	–	–	–	x	B: Lesefehler
–	–	x	x	–	–	–	–	–	–	–	–	–	–	–	x	C: allgemeiner Fehler
x	–	x	x	–	–	–	–	–	–	–	–	–	–	–	x	D: reserviert
–	x	x	x	–	–	–	–	–	–	–	–	–	–	–	x	E: reserviert
x	x	x	x	–	–	–	–	–	–	–	–	–	–	–	x	F: ungültiger Laufwerkswechsel

Tabelle 6. Jedes Gerät wird über zahlreiche Funktionen gesteuert

Das Betriebssystem ruft die Strategieroutine nur einmal während der Initialisierung des Treibers auf. Die Aufgabe der Prozedur ist es, die Adresse des Request Headers zu ermitteln und zwischenzuspeichern, so daß die Unterprogramme des Treibers immer auf diesen wichtigen Datenblock zugreifen können.

In Tabelle 5 finden Sie den typischen Aufbau eines Request Headers. Er wird im Listing nach dem Device Header mit der Adreßangabe »laenge« eingeleitet.

Der dritte und letzte Bestandteil des Gerätetreibers ist der sogenannte »Device Interrupt Handler«. Er übernimmt drei Aufgaben. Zuerst einmal enthält er eine Befehlstabelle, die einmal die Adressen der einzelnen Treiberfunktionen zur Verfügung stellt (im Listing mit der Adreßangabe »tabelle« eingeleitet).

Gerätetreiber-Request-Header

Offset	Länge	Name	Funktion
0	1 Byte	Länge eines Datensatzes	Länge des Request-Headers in Bytes
1	1 Byte	Einheitscode	Untereinheit
2	1 Byte	Funktionscode	siehe Tabelle 8
3	16 Byte	Status	Übernimmt Status der Funktion
5	8 Byte	Reserviert	

Tabelle 7. Das Statusfeld gibt an, ob eine Funktion erfolgreich durchgeführt wurde

Befehlscodes

Kommandocode	Funktion
0	Initialisierung
1	Prüfung Medienwechsel (nur Blocktreiber)
2	BIOS-Parameterblock erzeugen (nur Blocktreiber)
3	Ein-/Ausgabe-gesteuertes Lesen
4	Lesen von Gerät
5	Nicht zerstörendes Lesen (nur Zeichentreiber)
6	Lesestatus (nur Zeichentreiber)
7	Lesepuffer löschen (nur Zeichentreiber)
8	Schreiben
9	Schreiben mit Überprüfung
A	Schreibstatus (nur Zeichentreiber)
B	Schreibpuffer leeren (nur Zeichentreiber)
C	Ein-/Ausgabe-gesteuertes Schreiben

Tabelle 8. Die Fehlercodes ermöglichen eine Eingrenzung des Fehlerbereichs

Ein Gerätetreiber besteht aus zwei »Headers« und einem »Handler«

Des weiteren beinhaltet der Device Interrupt Handler zu allen Tabelleneinträgen die zugehörigen Routinen, die für einen reibungslosen Programmablauf verantwortlich sind. In diesem Fall bewirken die am Ende des Listings befindlichen Routinen nichts, außer daß sie wieder ins Hauptprogramm zurückkehren.

Zuletzt ist der Interrupt Handler noch dafür verantwortlich, den Befehlscode aus dem Request Header auszulesen, die entsprechende Funktion auszuführen und schließlich bei Auftreten eines Fehlers das Statusfeld zu ändern.

Die Kommandos, die der Gerätetreiber bereitstellt, zeigt Tabelle 6. Zu beachten ist, daß viele Funktionen nur für einen Treibertyp (zeichen- oder blockorientiert) gültig sind.

Nach der Ausführung eines Befehls erkennen Sie am Statusfeld, dessen Aufbau in Tabelle 7 ersichtlich ist, ob der Treiber eine Operation erfolgreich durchgeführt hat.

Eine hohe Anzahl von Kommandos vereinfacht die Arbeit mit Gerätetreibern

Bei einem Fehler läßt sich leicht feststellen, was zum Programmabbruch geführt hat. Tabelle 8 gibt Aufschluß über die Fehlerursache. Wenn Sie einen Treiber erfolgreich installiert haben, sprechen Sie ihn über die Funktionen des MS-DOS-Interrupt 21hex an. Welche Kommandos dabei Verwendung finden, hängt im wesentlichen davon ab, ob der Treiber einen Standardtreiber ersetzt oder als weiterer Treiber in das bestehende Computersystem hinzugefügt wird.

Zugegeben, die Thematik Gerätetreiber ist nicht ganz einfach zu verstehen, doch bieten sich Gerätetreiber als leistungsfähige Instrumente zur Vereinfachung der Kommunikation mit externen Hardware-Erweiterungen an und werden vom erfahrenen Programmierer entsprechend gerne genutzt.

(Markus Zietlow/wn)

```
[RU6]    1: ; Gerüst eines Standard-Gerätetreibers
[NX5]    2: ;    Geschrieben von Markus Zietlow
[9D6]    3: ;  (c) Redaktionsbüro Everts & Hagedorn
[P3Ø]    4: ;
[ZL4]    5: code      segment public 'code'
[PP2]    6: ;-Segmentanfang
[8A1]    7:           org     Øh
[WH1]    8: ;-Offset Ø
[P3Ø]    9: ;
[T95]   1Ø:           assume  cs:code, ds:code, es:code
[ZM3]   11: ;-Segmentzuweisungen
[PA4]   12: header:   dw      Øffffh, Øffffh
[EK4]   13: ;-Zeiger auf den Anfang des
[SK4]   14: ; nächsten Gerätetreibers
[DF1]   15:           dw      8ØØØh
[RU6]   16: ;-Bit 15=1: Treiber ist zeichenorientiert
[DT5]   17: ;-Bit 14=Ø: IOCTL wird unterstützt
[U76]   18: ;-Bit 13=Ø: Ohne Bedeutung für Zeichen-
[CC2]   19: ;           treiber
```

```
[HG3]   2Ø: ;-Bit 12=Ø: Reserviert
[GG3]   21: ;-Bit 11=Ø: Reserviert
[FG3]   22: ;-Bit 1Ø=Ø: Reserviert
[ME3]   23: ;-Bit  9=Ø: Reserviert
[LE3]   24: ;-Bit  8=Ø: Reserviert
[KE3]   25: ;-Bit  7=Ø: Reserviert
[HE3]   26: ;-Bit  6=Ø: Reserviert
[GE3]   27: ;-Bit  5=Ø: Reserviert
[VC4]   28: ;-Bit  4=Ø: Treiber nicht als
[VF3]   29: ; 'special' deklariert
[GE3]   3Ø: ;-Bit  3=Ø: Device kein
[AE2]   31: ; Uhrentreiber
[7N3]   32: ;-Bit  2=Ø: Treiber kein
[M22]   33: ; Null-Device
[XH4]   34: ;-Bit  1=Ø: Treiber ist nicht
[A74]   35: ; Standard-Ausgabegerät
[WH4]   36: ;-Bit  Ø=Ø: Treiber ist nicht
[764]   37: ; Standard-Eingabegerät
[YF2]   38:           dw      strategie
```

```
[Y46]   39: ;-Startadresse der Strategieroutine
[4L2]   4Ø:           dw      interrupt
[NL6]   41: ;-Anfangsadresse der Interrupt-Routine
[822]   42:           db      'DEVICE'
[Z43]   43: ;-Name des Treibers
[YG6]   44: ;Definition der Struktur des Request-
[EB1]   45: ;Blocks
[9E2]   46: laenge    equ     Ø
[ZB4]   47: ;-Länge des Blocks (1 Byte)
[ST3]   48: einheit   equ     laenge+1
[DK4]   49: ;-Nummer der Einheit (1 Byte)
[T14]   5Ø: kommando  equ     einheit+1
[5R3]   51: ;-Kommando-Code (1 Byte)
[C14]   52: status    equ     kommando+1
[B54]   53: ;-System-Status (2 Bytes)
[RZ3]   54: req_dos   equ     status+2
[1V3]   55: ;-Adresse des folgenden
[9K3]   56: ; Headers (DOS) (4 Bytes)
[G84]   57: req_user  equ     req_dos+4
[1V3]   58: ;-Adresse des folgenden
[XX3]   59: ; Headers (User) (4 Bytes)
[DC4]   6Ø: schluss   equ     req_user+5
[1K2]   61: ;-Ende des Blocks
[XL6]   62: ; Befehlstabelle für Treiberkommandos
[2C1]   63: tabelle:
[672]   64:           dw      initial
[665]   65: ;-Initialisierung des Treibers
[1X2]   66:           dw      media_check
[T64]   67: ;-Prüfung Medienwechsel
[P23]   68:           dw      bpb_erzeugen
[YF5]   69: ;-Aufbau des BIOS-Parameterblocks
[UF2]   7Ø:           dw      ioctl_ein
[995]   71: ;-Ein-/Ausgabe-gesteuertes Lesen
[D52]   72:           dw      eingabe
[Z81]   73: ;-Lesen
[4E2]   74:           dw      nz_lesen
[5U4]   75: ;-Nicht-zerstörendes Lesen
[6S2]   76:           dw      ein_status
[452]   77: ;-Lese-Status
[AN2]   78:           dw      ein_leeren
[VP3]   79: ;-Lesepuffer löschen
[X62]   8Ø:           dw      ausgabe
[UZ1]   81: ;-Schreiben
[GD2]   82:           dw      aus_test
[RØ5]   83: ;-Schreiben mit Überprüfen
[PT2]   84:           dw      aus_status
[ZT2]   85: ;-Schreib-Status
[UP2]   86:           dw      aus_leeren
[1W3]   87: ;-Schreibpuffer leeren
[ØG2]   88:           dw      ioctl_aus
[GZ5]   89: ;-Ein-/Ausgabe-gesteuertes Schreiben
[P3Ø]   9Ø: ;
[MD7]   91: ; Strategie-Routine des Einheitentreibers,
[2V6]   92: ;die die Adresse der Anforderungstabelle
[H35]   93: ;temporär zwischenspeichert
[P3Ø]   94: ;
[793]   95: strategie proc    far
[2F3]   96: ;-Anfang der Prozedur
[C95]   97:           mov     cs:word ptr doszeiger, bx
[BR2]   98: ;-Adresse ablegen
[S85]   99:           mov     cs:word ptr doszeiger+2, es
[CW4]   1ØØ: ;-Segment zwischenspeichern
[3PØ]   1Ø1:          ret
[TV4]   1Ø2: ;-Rückkehr ins Hauptprogramm
[BE2]   1Ø3: strategie endp
[R63]   1Ø4: ;-Ende der Prozedur
[W93]   1Ø5: doszeiger dw     Øh,Øh
[KR7]   1Ø6: ;-Zwischenspeicher für Adresse:Anforderungs
             -
[VC1]   1Ø7: ;Tabelle
[DC3]   1Ø8: interrupt proc   far
[2F3]   1Ø9: ;-Anfang der Prozedur
[MF1]   11Ø:          push    si
[EA5]   111: ;-Alle wichtigen Register mit den
[VP5]   112: ;Betriebssystemdaten durch  PUSH-
[7L3]   113: ;Operationen  sichern
[XF1]   114:          push    ax
[1G1]   115:          push    cx
[3G1]   116:          push    dx
[PE1]   117:          push    di
[7F1]   118:          push    bp
[MF1]   119:          push    ds
[PF1]   12Ø:          push    es
[ZF1]   121:          push    bx
[RC5]   122:          lds     bx,cs:dword ptr doszeiger
[MH3]   123: ;-Den DOS-Zeiger laden
[G23]   124:          assume  ds:nothing
[9N1]   125:          mov     ax,Ø
[UF3]   126: ;-Fehler-Code löschen
[RR5]   127:          mov     al,ds:byte ptr kommando[b
          x]
[WH2]   128: ;-Kommando laden
[ØX3]   129:          mov     si,offset tabelle
[7Z3]   13Ø: ; und Adresse der Tabelle
[ØR1]   131:          add     si,ax
[UL3]   132: ; als Wort definierten
[ØR1]   133:          add     si,ax
```

```
[YC3]   134: ; Funktion ermitteln
[PN1]   135:          cmp     al,11
[235]   136: ;-Falls Befehlscode zulässig,
[WZ3]   137: ; Sprung zu 'befehl_ok',
[ZL2]   138:          jng     befehl_ok
[N52]   139:          mov     ax,8ØØ3h
[UH5]   14Ø: ; ansonsten Fehlercode setzen und
[F92]   141: ;unterbrechen
[5T1]   142:          jmp     exit
[573]   143: befehl_ok: push  cs
[736]   144: ;-Datensegment neu festlegen und zum
[X25]   145: ;gewünschten Befehl springen
[ED1]   146:          pop     ds
[FK2]   147:          assume  ds:code
[BU1]   148:          jmp     [si]
[YVØ]   149: exit:
[G23]   15Ø:          assume  ds:nothing
[SR5]   151: ;-Den Status 'Beendet' setzen und im
[WZ6]   152: ;Status-Byte des Request-Headers ablegen
[4C2]   153:          or      ah,ØØØØØØØ1b
[RC5]   154:          lds     bx,cs:dword ptr doszeiger
[UL5]   155:          mov     ds:word ptr status[bx],ax
[TD1]   156:          pop     bx
[FY5]   157: ;-Alle gesicherten Register mit den
[ZN7]   158: ; Betriebssystemdaten durch POP-Operationen
[S43]   159: ; wiederherstellen
[GD1]   16Ø:          pop     es
[ED1]   161:          pop     ds
[1D1]   162:          pop     bp
[GC1]   163:          pop     di
[XD1]   164:          pop     dx
[VD1]   165:          pop     cx
[RD1]   166:          pop     ax
[ED1]   167:          pop     si
[3PØ]   168:          ret
[TV4]   169: ;-Rückkehr ins Hauptprogramm
[F94]   17Ø: ;Einschaltmeldung für den
[KU3]   171: ;Initialisierungsteil
[113]   172: meldung  db      13,1Ø
[DK8]   173:          db      "Beispiel-Gerätetreiber v
          on Markus Zietlow "
[NL6]   174:          db      "aus der DOS Internationa
          l geladen"
[YP1]   175:          db      13,1Ø,"$"
[4C7]   176: ; Anfang des Initialisierungsteils, der bei
[8P5]   177: ;der Installation automatisch vom
[RL5]   178: ;Betriebssystem aufgerufen wird
[KX3]   179: initial:  lea     ax,e_prg
[LD4]   18Ø: ;gibt Ende des Programms an
[EP4]   181:          mov     es:word ptr[bx]+14,ax
[CP4]   182:          mov     es:word ptr[bx]+16,cs
[KY3]   183:          mov     dx,offset meldung
[TT3]   184: ;-DX=Anfang des Textes
[2M1]   185:          mov     ah,9
[BT3]   186: ;-Mit der DOS-Funktion 9
[3K1]   187:          int     21h
[4H3]   188: ; die Meldung ausgeben
[9N1]   189:          mov     ax,Ø
[244]   19Ø: ;-Fehler-Flag löschen und
[5T1]   191:          jmp     exit
[KU4]   192: ; Rückkehr ins Hauptprogramm
[P3Ø]   193: ;
[WL3]   194: media_check: mov   ax,Ø
[UR4]   195: ;-Fehler-Flag löschen und ins
[914]   196: ; Hauptprogramm zurück
[5T1]   197:          jmp     exit
[PU3]   198: bpb_erzeugen: mov   ax,Ø
[5T1]   199:          jmp     exit
[XC3]   2ØØ: ioctl_ein: mov     ax,Ø
[5T1]   2Ø1:          jmp     exit
[D13]   2Ø2: eingabe:  mov     ax,Ø
[5T1]   2Ø3:          jmp     exit
[Y93]   2Ø4: nz_lesen: mov     ax,Ø
[5T1]   2Ø5:          jmp     exit
[4M3]   2Ø6: ein_status: mov   ax,Ø
[5T1]   2Ø7:          jmp     exit
[8H3]   2Ø8: ein_leeren: mov   ax,Ø
[5T1]   2Ø9:          jmp     exit
[X23]   21Ø: ausgabe:  mov     ax,Ø
[5T1]   211:          jmp     exit
[A93]   212: aus_test: mov     ax,Ø
[5T1]   213:          jmp     exit
[MN3]   214: aus_status: mov   ax,Ø
[5T1]   215:          jmp     exit
[SK3]   216: aus_leeren: mov   ax,Ø
[5T1]   217:          jmp     exit
[K93]   218: ioctl_aus: mov    ax,Ø
[5T1]   219:          jmp     exit
[N31]   22Ø: e_prg:
[HH2]   221: interrupt endp
[R63]   222: ;-Ende der Prozedur
[BW1]   223: code      ends
[ØV3]   224: ;-Ende des Code-Segments
[4MØ]   225:          end
[LT4]   226: ;-Ende des Assemblerprogramms
```

Listing. Jede Struktur eines Gerätetreibers ist grundsätzlich auf diese Weise aufgebaut

Der geklonte Prozessor

NEC V20 und V30 im Detail

In vielen PCs versieht anstelle eines 8088- oder 8086-Prozessors eine Zentraleinheit vom Typ V20 oder V30 des Herstellers NEC seinen Dienst. Gegenüber dem einfachen 8088/8086 sind die NEC-Prozessoren bei gleicher Taktfrequenz erheblich schneller und verfügen zusätzlich über einen erweiterten Befehlssatz.

Die NEC-Prozessoren V20 und V30 sind vollständig pin- und softwarekompatibel mit den Intel-Prozessoren 8088 und 8086. Aus diesem Grund läßt sich der 8088/8086 in jedem PC problemlos gegen einen NEC-Prozessor austauschen, ohne daß irgendwelche Probleme mit vorhandener Hard- oder Software auftreten.
Dabei ist der V20 der Austauschprozessor für den 8088, während der V30 dem 8086 entspricht. Die Prozessoren V20 und V30 unterscheiden sich, wie auch die Intel-Prozessoren 8088 und 8086, lediglich durch die Breite des Datenbusses, der beim V20 8 Bit und beim V30 16 Bit breit ist.
Der erste wesentliche Unterschied zwischen den »normalen« 8088/8086 und den NEC-Prozessoren ist der deutlich geringere Stromverbrauch. Während die 5-MHz-Ausführung des 8086 1,7 Watt benötigt, begnügt sich der adäquate V30-Prozessor mit nur 0,15 Watt. Diese enorme Ersparnis resultiert daraus, daß die NEC-Prozessoren vollständig in CMOS-Technik hergestellt werden. Der Fairneß halber soll nicht verschwiegen werden, daß es inzwischen auch eine CMOS-Version des 8088/8086 gibt, die einen ähnlich niedrigen Stromverbrauch aufweist. Im gewöhnlichen PC-Betrieb macht sich der unterschiedliche Stromverbrauch allerdings nicht bemerkbar. Lediglich bei Laptops spielen die NEC-Prozessoren diesen Vorteil aus.
Des weiteren bestechen die V20/V30-Prozessoren durch eine höhere Verarbeitungsgeschwindigkeit bei gleicher Taktfrequenz. Diesen Vorteil erzielten die NEC-Konstrukteure, indem sie zahlreiche Befehle, die beim 8088/8086 noch als Microcode gespeichert sind, als Hardware-Rechenwerk verdrahteten. Während ein im Microcode (eine Art Miniaturprogramm im Innern des Prozessors) vorliegender Befehl zur Ausführung eine Vielzahl von Taktzyklen benötigt, sind für die gleiche Aufgabe bei hardwareseitig codierten Befehlen wesentlich weniger Taktzyklen notwendig.
Aus diesem Grund betrifft die Geschwindigkeitssteigerung aber auch nur bestimmte Befehle, die dann aber zum Teil in einem Viertel der Zeit ausgeführt werden. Insbesondere die Befehle für die Multiplikation und die Division sind davon betroffen. Andere Befehle hingegen werden nur um 30 Prozent beschleunigt, einige wenige erfahren überhaupt keinen Geschwindigkeitszuwachs, weil sie schon beim 8088/8086 optimiert sind.

Die NEC-Prozessoren arbeiten bei gleicher Taktgeschwindigkeit deutlich schneller

Damit sind aber noch nicht alle Vorteile der NEC-Prozessoren aufgezählt, denn sie sind nicht nur schneller als der gewöhliche 8088/8086, sondern auch beträchtlich leistungsfähiger. So verfügen sie zum Beispiel über einen erheblich erweiterten Befehlsvorrat.
Außerdem sind einige Fehler des 8088/8086 ausgemerzt worden. Einer dieser Fehler des 8088/8086 tritt zwar nur unter sehr seltenen Umständen auf, ist dafür aber eindeutig nachvollziehbar: Bei String-Operationen, die Sie zusammen mit einem Segment-Override (also zum Beispiel »CS:«) und dem Rep-(Wiederholungs-) Präfix verwenden, wird die Ausführung der String-Operation nicht fortgeführt, wenn zwischenzeitlich ein Hardware-Interrupt auftritt. Obwohl das als Zähler fungierende CX-Register dann noch nicht den Wert 0 enthält, beendet der Prozessor die Ausführung der Schleife nach der Rückkehr vom Hardware-Interrupt. Dieser Fehler ist beim V20/V30-Prozessor behoben.
Die zusätzlichen Befehle des V20/V30 gliedern sich in zwei Gruppen: Zum einen beherrschen die NEC-Prozessoren alle zusätzlichen Befehle des 80286 im »Real Mode«, zum anderen gibt es noch einige NEC-spezifische Befehle. Tabelle 1 zeigt eine Auflistung der 80286-kompatiblen Anweisungen, die die NEC-Prozessoren verstehen.
Leider hat NEC den 80286-Befehlen aber andere Namen gegeben, als die von Intel verwendeten Mnemonics (vielleicht um Copyright-Problemen aus dem Weg zu gehen). So heißt beispielsweise der Befehl, der alle Register auf den Stapel legt, bei Intel »pusha« und bei NEC »push r«. Dabei handelt es sich jedoch in beiden Fällen um die gleiche Anweisung, die auch auf die gleiche Weise codiert wird.
Wenn Sie nicht gerade in Assembler programmieren, wirkt sich das für Sie kaum störend aus; Sie können die zusätzlichen Befehle trotzdem nutzen. Viele Hochsprachen-Compiler erlauben nämlich die Wahl des Prozessors, für den das Programm optimiert werden soll. Wählen Sie hier den 80186 oder den 80286 aus, dann verwendet der Compiler – wann immer es günstiger ist – die zusätz-

Befehl	Funktion
push n	Lege Wert n auf den Stapel
push r	Lege alle acht Rgister auf den Stapel
pop r	Hole alle acht Register vom Stapel
mul reg,nn	Multipliziere Register mit nn
mul (nnnn),nn	Multipliziere den Inhalt von nnnn mit nn
shl reg,n	Schiebe Register n-mal nach links
shr reg,n	Schiebe Register n-mal nach rechts
rol reg,n	Rotiere Register n-mal links
ror reg,n	Rotiere Register n-mal rechts
rolc reg,n	Rotiere Register n-mal links mit Carry
rorc reg,n	Rotiere Register n-mal rechts mit Carry
chkind reg,adr	Überprüfe, ob Index-Wert innerhalb des erlaubten Bereichs
outm	Gib String an I/O-Port aus
inm	Lies String von I/O-Port ein
prepare nnnn,nn	Richte zweiten Stack ein
dispose	Zurück zum alten Stack

Tabelle 1. Der V20/V30 versteht alle zusätzlichen Befehle des 80286, abgesehen von jenen für den »Protected Mode«. Allerdings ist die Bezeichnung identischer Befehle bei NEC und Intel unterschiedlich

lichen 80286-Befehle. Da der V20/V30 alle diese Befehle versteht (denn die Codierung dieser Anweisungen entspricht vollständig der des 80286), läuft das Compilat auch auf einem Rechner mit V20/V30-Prozessor. Auf einem PC mit einem 8088/8086-Prozessor ist das entstandene Programm dann allerdings nicht mehr lauffähig.

Neben diesen 80286-kompatiblen Befehlen kennt der V20/V30-Prozessor zusätzlich noch einige NEC-spezifische Befehle. Tabelle 2 zeigt eine Auflistung dieser Anweisungen. Diese sehr nützlichen Zusatzbefehle lassen sich in einer Hochsprache (zum Beispiel Turbo Pascal) aber allenfalls innerhalb einer Inline-Prozedur verwenden, denn sie sind weder beim 80186, 80286 noch beim 80386 vorhanden.

Gewöhnliche Assembler kennen die zusätzlichen Befehle der NEC-Prozessoren nicht

Auch gewöhnliche Assembler für den PC kennen die NEC-spezifischen Zusatzbefehle nicht. Für die 80286-kompatiblen Befehle können Sie sich behelfen, indem Sie beim MASM die Direktive ».286« einfügen. Dann dürfen Sie diese Befehle verwenden, aber unter den Intel-üblichen Namen, wie zum Beispiel »»pusha« statt »push r« und so weiter. Für die NEC-spezifischen Befehle müssen Sie sich entsprechende Makros schreiben.

Lediglich der von NEC angebotene V20/V30-Assembler für den IBM-PC unter MS-DOS, der die Typenbezeichnung »RAPCD-I5DD-70116« trägt, beherrscht die Codierung der zusätzlichen V20/V30-Befehle ohne weiteres und in der ursprünglichen NEC-Schreibweise.

Wer sich eingehender mit der Programmierung des V20/V30-Prozessors beschäftigen möchte, dem sei das Buch »Microprocessors and Peripherals Data Book« empfohlen. Es ist direkt bei NEC oder einem Vertragshändler erhältlich und erklärt die Funktionen und Befehle des V20/V30 recht ausführlich, allerdings in englischer Sprache.

Nicht nur wegen der zusätzlichen Assemblerbefehle bietet der V20/V30 dem Programmierer und auch dem Anwender bedeutende Vorteile. Als »Bonbon« haben die Entwickler diesen Mikroprozessor zusätzlich zu dem gewöhnlichen, auch »Native Mode« genannten 8088/8086-Modus mit einen 8080-kompatiblen Modus ausgestattet.

In diesem »Emulation Mode« emuliert ein V20/V30 den vollständigen Befehlssatz des 8080-Prozessors, für den das seinerzeit weltweit verbreitete Betriebssystem CP/M geschrieben wurde. Dies versetzt Sie in die Lage, gewöhnliche CP/M-Software auf Ihrem PC mit V20/V30-Prozessor ablaufen zu lassen.

Notwendig ist dazu außer einem V20/V30 nur noch ein speziell an die PC-Umgebung angepaßtes CP/M, das bereits von einigen Firmen vertrieben wird. Sogar im Fundus der Public-Domain-Software und Shareware finden sich entsprechende CP/M-Emulatoren. Allerdings handelt es sich eben um einen 8080-Emulationsmodus, und die erweiterten Befehle des Z80-Prozessors sind ihm fremd. Daher laufen nur solche CP/M-Programme auf Ihrem PC, die keine Z80-spezifischen Anweisungen enthalten.

Zum Umschalten zwischen dem normalen und dem Emulationsmodus gibt es einige spezielle Befehle. Mit »brkem n« (»break for emulation«) schalten Sie den V20/V30-Prozessor in den 8080-Emulationsmodus. Nach der Ausführung dieses Befehls werden das Flag-Register sowie der Befehlszähler und das Codesegment auf dem Stapel abgelegt. Nun lädt der Mikroprozessor den Befehlszähler und das Codesegmentregister mit dem Inhalt des Interrupt-Vektors »n«.

Der an dieser Adresse beginnende Code wird dann im 8080-Emulationsmodus ausgeführt. In diesem Modus übernehmen die gewöhnlichen Register des V20/V30 die Aufgaben der 8080-Register (Tabelle 3). Durch die Verwendung zweier verschiedener Stapelzeiger für den 8080-Stack und den 8088/8086-Stack wird eine Datenkollision vermieden, und es gibt keine Probleme beim Umschalten zwischen den beiden Modi. Im 8080-Modus sind die Segmentregister natürlich unveränderlich, weil der 8080 nur einen 64 KByte großen Adreßbereich ansprechen kann. Daher verwendet der V20/V30 für die Befehlsausführung automatisch das CS-Register, dessen Inhalt durch den entsprechenden Interrupt-Vektor beim Aufruf des Emulationsmodus bestimmt wurde. Bei allen Zugriffen auf Daten wird das DS-Register verwendet, dessen Wert Sie noch im 8086-Modus entsprechend einstellen müssen.

Der Befehl »calln n« führt Sie vom 8080-Modus in den 8086-Modus zurück. Der V20/V30 setzt dadurch den Programmablauf bei der durch den Interrupt-Vektor »n« festgelegten Adresse fort. Zurück zum »Emulation Mode« gelangen Sie durch den Assemblerbefehl »reti«. Der Umschaltvorgang zwischen den beiden Modi läuft also in etwa wie ein Software-Interrupt ab. Am Ende des 8080-Programms steht sinnvollerweise der Befehl »retem«, der in den 8088/8086-Modus zurückführt, ohne dabei die 8080-Register und 8080-Statusbits zu retten.

(Martin Hepp/zi)

8080	V20/V30
A	AL
B	CH
C	CL
D	DH
E	DL
H	BH
L	BL
SP	BP
PC	IP
Statuskennzeichen	
Carry	C
Zero	Z
Sign	S
Parity	P
Aux-Carry	AC

Tabelle 3. Im 8080-Emulationsmodus übernehmen die Register des V20/V30 die Aufgaben der 8080-Register

Befehl	Funktion
ins	Füge Bitfeld ein
ext	Entferne Bitfeld
add4s	Addiere gepackten Dezimal-String
sub4s	Subtrahiere einen gepackten Dezimal-String vom anderen
cmp4s	Vergleiche zwei gepackte Dezimal-Strings
rol4	Rotiere eine BCD-Stelle links
ror4	Rotiere eine BCD-Stelle rechts
test1	Teste angegebenes Bit
not1	Invertiere angegebenes Bit
clr1	Lösche angegebenes Bit
set1	Setze angegebenes Bit
repc	Führe weitere Befehle aus, bis CY-Flag gelöscht
repnc	Führe weitere Befehle aus, bis CY-Flag gesetzt
fpo2	Rufe mathematischen Coprozessor auf

Tabelle 2. Zusätzlich zu den 80286-Befehlen kennt ein Prozessor vom Typ V20 oder V30 eine Reihe von NEC-spezifischen Befehlen

Info:
»Microcomputers and Peripherals Data Book«, NEC Electronics Europe GmbH, 4000 Düsseldorf 30

Die Assemblerbefehle des 8086/8088

Maschinensprache ist für viele PC-Besitzer ein Buch mit sieben Siegeln. Dabei läßt sich mit einem leistungsfähigen Assembler ähnlich komfortabel programmieren wie mit einer höheren Sprache. Nur die Befehle und Register lassen sich – zugegebenermaßen – schwer im Kopf behalten. Deshalb finden Sie in dieser DOS-Extra-Ausgabe Referenztabellen, die das ewige Nachschlagen in vielen unterschiedlichen Handbüchern überflüssig machen.

Die meisten PC-Besitzer programmieren zwar in GW-Basic und Turbo Pascal, doch eigentlich läßt sich der Prozessor des PC (und das gilt auch für jeden anderen Computer) nur in Maschinencode programmieren. Bei GW-Basic übersetzt deshalb der Interpreter ein Programm während des Ablaufs Befehl für Befehl in Maschinencode. Turbo-Pascal-Programme werden von einem Compiler zuerst vollständig in Maschinencode übertragen, bevor sie überhaupt ablauffähig sind.

Wenn Sie hingegen direkt in Maschinensprache programmieren, so erhalten Sie wesentlich schnellere Programme. Doch Maschinencode besteht nur aus endlosen Zahlenkolonnen, die für den Menschen meist völlig unübersichtlich sind.

Aus diesem Grund haben sich die Entwickler von Mikroprozessoren zu allen Maschinencodebefehlen kurze, aber verständliche Bezeichnungen (zum Beispiel »ADD« für »addieren«, »INT« für »unterbrechen« oder »OUT« für »ausgeben«) einfallen lassen. Diese Kurzbezeichnungen – auch Assemblerbefehle genannt – vereinfachen das Programmieren der Prozessoren erheblich. Um die Assemblerbefehle jedoch wieder in Maschinencode umzuformen, ist der Einsatz eines Übersetzungsprogramms, eben eines Assemblers, erforderlich.

Tabelle 1 zeigt alle Assemblerbefehle des 8086/8088-Prozessors jeweils mit der ausführlichen englischen Bezeichnung und der direkten, deutschen Übersetzung. Die englischen Bezeichnungen lassen sich viel leichter merken als die Abkürzungen. Tabelle 2 erläutert die grundlegende Funktion jedes Assemblerbefehls, und Tabelle 3 listet die Assemblerbefehle alphabetisch geordnet mit dem passenden Maschinencode und der genauen Bedeutung auf. Diese Tabelle ist besonders hilfreich bei der Suche nach einem bestimmten Assemblerbefehl und beim Übersetzen des Befehls in Maschinencode.

Tabelle 4 führt dagegen die Assemblerbefehle numerisch nach dem Maschinencode geordnet auf. Sie können mit dieser Tabelle kurze Maschinencode-Listings sehr schnell von Hand übersetzen, wenn Ihnen gerade kein Disassembler zur Verfügung steht.

Die Bedeutungen und Codierungen der im Maschinencode verwendeten Parameter und Abkürzungen gehen aus den Tabellen 5 bis 11 hervor.

(ma/tr)

Die 8086/8088-Befehle in Englisch und Deutsch

Abkürz.	Englisch	Deutsch
AAA	ASCII adjust after addition	ASCII-Anpassung nach einer Addition
AAD	ASCII adjust before divide	ASCII-Anpassung vor einer Division
AAM	ASCII adjust after multiply	ASCII-Anpassung nach einer Multiplikation
AAS	ASCII adjust after subtract	ASCII-Anpassung nach einer Subtraktion
ADC	Add with carry	Addiere mit Übertrag-Statusbit
ADD	Add	Addiere
AND	And	Und
CALL	Call	Rufe auf
CBW	Convert byte to word	Wandle Byte in Wort um
CLC	Clear carry flag	Lösche Übertrag-Statusbit
CLD	Clear direction flag	Lösche Richtung-Statusbit
CLI	Clear interrupt flag	Lösche Unterbrechung-Statusbit
CMC	Complement carry flag	Invertiere Übertrag-Statusbit
CMP	Compare	Vergleiche
CMPS	Compare string	Vergleiche Zeichenkette
CMPSB	Compare string bytewise	Vergleiche Zeichenkette byteweise
CMPSW	Compare string wordwise	Vergleiche Zeichenkette wortweise
CWD	Convert word to double word	Wandle Wort in Doppelwort um
DAA	Decimal adjust after addition	Dezimale Anpassung nach einer Addition
DAS	Decimal adjust after subtract	Dezimale Anpassung nach einer Subtraktion
DEC	Decrement	Vermindere
DIV	Divide (unsigned)	Dividiere (vorzeichenlos)
ESC	Escape (to external device)	Entweiche (zu externer Einheit)
HLT	Halt	Halt
IDIV	Integer divide (signed)	Ganzzahlige Division (vorzeichenbehaftet)
IMUL	Integer multiply (signed)	Ganzzahlige Multiplikation (vorzeichenbehaftet)
IN	Input from	Lese ein von
INC	Increment	Erhöhe
INT	Interrupt	Unterbreche
INTO	Interrupt if overflow	Unterbreche, wenn Überlauf
IRET	Interrupt return	Kehre von Unterbrechungsprogramm zurück
JA	Jump if above	Springe, wenn darüber
JAE	Jump if above or equal	Springe, wenn darüber oder gleich
JB	Jump if below	Springe, wenn darunter
JBE	Jump if below or equal	Springe, wenn darunter oder gleich
JCXZ	Jump if CX equal zero	Springe, wenn CX gleich 0
JE	Jump if equal	Springe, wenn gleich
JG	Jump if greater	Springe, wenn größer
JGE	Jump if greater or equal	Springe, wenn größer oder gleich
JL	Jump if less	Springe, wenn kleiner
JLE	Jump if less or equal	Springe, wenn kleiner oder gleich
JMP	Jump	Springe
JNA	Jump if not above	Springe, wenn nicht darüber
JNAE	Jump if not above and not equal	Springe, wenn nicht darüber und nicht gleich
JNB	Jump if not below	Springe, wenn nicht darunter
JNBE	Jump if not below and not equal	Springe, wenn nicht darunter und nicht gleich
JNE	Jump if not equal	Springe, wenn nicht gleich
JNG	Jump if not greater	Springe, wenn nicht größer
JNGE	Jump if not greater and not equal	Springe, wenn nicht größer und nicht gleich
JNL	Jump if not less	Springe, wenn nicht kleiner
JNLE	Jump if not less and not equal	Springe, wenn nicht kleiner und nicht gleich
JNO	Jump if no overflow	Springe, wenn kein Überlauf
JNP	Jump if no parity	Springe, wenn keine Parität
JNS	Jump if no sign	Springe, wenn kein Vorzeichen (positiv)
JNZ	Jump if not zero	Springe, wenn nicht 0
JO	Jump if overflow	Springe, wenn Überlauf
JP	Jump if parity	Springe, wenn Parität
JPE	Jump if parity even	Springe, wenn Parität gerade
JPO	Jump if parity odd	Springe, wenn Parität ungerade
JS	Jump if sign	Springe, wenn Vorzeichen (negativ)
JZ	Jump if zero	Springe, wenn 0
LAHF	Load AH with flags	Lade AH mit den Statusbits
LDS	Load pointer using DS	Lade Zeiger unter Verwendung von DS
LEA	Load effective address to register	Lade effektive Adresse in Register
LES	Load pointer using ES	Lade Zeiger unter Verwendung von ES

Abkürz.	Englisch	Deutsch
LOCK	Lock Bus	Verriegle den Bus
LODS	Load string to AL/AX	Lade Zeichenkette in AL/AX
LODSB	Load string bytewise to AL	Lade Zeichenkette byteweise in AL
LODSW	Load string wordwise to AX	Lade Zeichenkette wortweise in AX
LOOP	Loop CX times	Führe Schleife CX-mal aus
LOOPE	Loop CX times while equal	Führe Schleife CX-mal aus solange gleich
LOOPNE	Loop CX times while not equal	Führe Schleife CX-mal aus solange ungleich
LOOPNZ	Loop CX times while not zero	Führe Schleife CX-mal aus solange nicht 0
LOOPZ	Loop CX times while zero	Führe Schleife CX-mal aus solange 0
MOV	Move	Verschiebe
MOVS	Move string	Verschiebe Zeichenkette
MOVESB	Move string bytewise	Verschiebe Zeichenkette byteweise
MOVESW	Move string wordwise	Verschiebe Zeichenkette wortweise
MUL	Muliply (unsigned)	Multipliziere (vorzeichenlos)
NEG	Negate	Negiere
NOP	No operation	Keine Operation
NOT	Invert	Invertiere
OR	Or	Oder
OUT	Output to	Gebe aus nach
POP	Pop	Hole
POPF	Pop flags	Hole Statusbits
PUSH	Push	Schiebe
PUSHF	Push flags	Schiebe Statusbits
RCL	Rotate through carry left	Rotiere durch Übertrag-Statusbit nach links
RCR	Rotate through carry right	Rotiere durch Übertrag-Statusbit nach rechts
REP	Repeat CX times	Wiederhole CX-mal
REPE	Repeat CX times while equal	Wiederhole CX-mal solange gleich
REPNE	Repeat CX times while not equal	Wiederhole CX-mal solange nicht gleich
REPNZ	Repeat CX times while not zero	Wiederhole CX-mal solange nicht gleich 0
REPZ	Repeat CX times while zero	Wiederhole CX-mal solange 0
RET	Return from procedure	Kehre aus Prozedur zurück
ROL	Rotate left	Rotiere nach links
ROR	Rotate right	Rotiere nach rechts
SAHF	Store AH into flags	Speichere AH in die Statusbits
SAL	Shift arithmetic left	Schiebe arithmetisch nach links
SAR	Shift arithmetic right	Schiebe arithmetisch nach rechts
SBB	Subtract with borrow	Subtrahiere mit Ausborgen
SCAS	Scan string	Taste Zeichenkette ab
SCASB	Scan string bytewise	Taste Zeichenkette byteweise ab
SCASW	Scan string wordwise	Taste Zeichenkette wortweise ab
SEG	Segment override	Überschreibe Segment
SHL	Shift logical left	Schiebe logisch nach links
SHR	Shift logical right	Schiebe logisch nach rechts
STC	Set carry flag	Setze Übertrag-Statusbit
STD	Set direction flag	Setze Richtung-Statusbit
STI	Set interrupt flag	Setze Unterbrechung-Statusbit
STOS	Store string from AL/AX	Speichere Zeichenkette aus AL/AX
STOSB	Store string bytewise from AL	Speichere Zeichenkette byteweise aus AL
STOSW	Store string wordwise from AX	Speichere Zeichenkette wortweise aus AX
SUB	Subtract	Subtrahiere
TEST	Test	Teste
WAIT	Wait for test	Warte auf Test-Signal
XCHG	Exchange	Tausche aus
XLAT	Translate byte to AL	Setze Byte nach AL um
XOR	Exclusive or	Exklusiv-Oder

Tabelle 1. Die Befehle des 8086/8088 mit englischer Bezeichnung und genauer deutscher Übersetzung

Die Befehle des 8086/8088 und ihre Funktion

Bezeichnung	Funktion
AAA	Korrigiere Ergebnis nach einer BCD-Addition
AAD	Korrigiere Bruch vor einer BCD-Division
AAM	Korrigiere Ergebnis nach einer BCD-Multiplikation
AAS	Korrigiere Ergebnis nach einer BCD-Subtraktion
ADC	Addiere mit Verwendung des Übertrag-Statusbits
ADD	Addiere ohne Übertrag
AND	Führe logische UND-Verknüpfung aus
CALL	Rufe Unterprogramm auf
CBW	Wandle ein Registerbyte in ein Wort um
CLC	Lösche das Übertrag-Statusbit
CLD	Lösche das Richtung-Statusbit
CLI	Lösche das Unterbrechung-Statusbit
CMC	Invertiere das Übertrag-Statusbit
CMP	Führe einen Vergleich aus
CMPS	Vergleiche zwei Zeichenketten
CMPSB	Vergleiche zwei Byte von Zeichenketten
CMPSW	Vergleiche zwei Worte von Zeichenketten
CWD	Wandle ein Registerwort in ein Doppelwort um
DAA	Korrigiere das Ergebnis einer BCD-Addition
DAS	Korrigiere das Ergebnis einer BCD-Subtraktion
DEC	Zähle einen Wert um 1 herab
DIV	Führe eine vorzeichenlose Division aus
ESC	Übergib einen Wert an eine externe Einheit
HLT	Halte den Prozessor an
IDIV	Führe eine vorzeichenbehaftete Division aus
IMUL	Führe eine vorzeichenbehaftete Multiplikation aus
IN	Lies einen Wert über einen Port
INC	Erhöhe einen Wert um 1
INT	Löse einen Interrupt aus
INTO	Löse bei einem Überlauf einen Interrupt aus
IRET	Kehre aus einer Interrupt-Routine zurück
JA	Verzweige, wenn größer als Null (entspricht JNBE)
JAE	Verzweige, wenn größer oder gleich Null (entspricht JNB)
JB	Verzweige, wenn kleiner als Null (entspricht JNAE)
JBE	Verzweige, wenn darunter oder gleich (entspricht JNA)
JCXZ	Verzweige, wenn Inhalt von CX gleich 0
JE	Verzweige, wenn gleich (entspricht JZ)
JG	Verzweige, wenn größer (entspricht JNLE)
JGE	Verzweige, wenn größer oder gleich (entspricht JNL)
JL	Verzweige, wenn kleiner (entspricht JNGE)
JLE	Verzweige, wenn kleiner oder gleich (entspricht JNG)
JMP	Verzweige
JNA	Verzweige, wenn nicht darüber (entspricht JBE)
JNAE	Verzweige, wenn nicht darüber und nicht gleich (entspricht JB)
JNB	Verzweige, wenn nicht darunter (entspricht JAE)
JNBE	Verzweige, wenn nicht darunter und nicht gleich (entspricht JA)
JNE	Verzweige, wenn nicht gleich (entspricht JNZ)
JNG	Verzweige, wenn nicht größer (entspricht JLE)
JNGE	Verzweige, wenn nicht größer und nicht gleich (entspricht JL)
JNL	Verzweige, wenn nicht kleiner (entspricht JGE)
JNLE	Verzweige, wenn nicht kleiner und nicht gleich (entspricht JG)
JNO	Verzweige, wenn kein Überlauf
JNP	Verzweige, wenn keine Parität (entspricht JPO)
JNS	Verzweige, wenn positiv
JNZ	Verzweige, wenn nicht 0 (entspricht JNE)
JO	Verzweige, wenn Überlauf
JP	Verzweige, wenn Parität (entspricht JPE)
JPE	Verzweige, wenn Parität gerade (entspricht JP)
JPO	Verzweige, wenn Parität ungerade (entspricht JNP)
JS	Verzweige, wenn negativ
JZ	Verzweige, wenn 0 (entspricht JE)
LAHF	Lade die 8 Statusbits in das AH-Register
LDS	Lade DS-Register und weiteres Register
LEA	Berechne und lade effektive Adresse
LES	Lade ES-Register und weiteres Register
LOCK	Aktiviere Lock-Signal für nachfolgenden Befehl
LODS	Lade Zeichenkette
LODSB	Lade Zeichenkette byteweise
LODSW	Lade Zeichenkette wortweise
LOOP	Wiederhole Schleife bis Zähler CX gleich 0
LOOPE	Wiederhole Schleife bis ungleich oder Zähler CX gleich 0
LOOPNE	Wiederhole Schleife bis gleich oder Zähler CX gleich 0
LOOPNZ	Wiederhole Schleife bis gleich 0 oder Zähler CX gleich 0
LOOPZ	Wiederhole Schleife bis ungleich 0 oder Zähler CX gleich 0
MOV	Übertrage Zeichen
MOVS	Übertrage Zeichenkette
MOVESB	Übertrage Zeichenkette byteweise
MOVESW	Übertrage Zeichenkette wortweise
MUL	Führe vorzeichenlose Multiplikation aus
NEG	Negiere (2er-Komplement bilden)
NOP	Führe keine Operation aus
NOT	Invertiere (1er-Komplement bilden)

Die Befehle des 8086/8088 und Ihre Funktion

Bezeich.	Funktion
OR	Führe logische OR-Verknüpfung aus
OUT	Gib über Port aus
POP	Lies Wort vom Stack
POPF	Lies Statusbits vom Stack
PUSH	Lege Zeichen auf Stack
PUSHF	Lege Statusbits auf Stack
RCL	Rotiere durch Übertrag-Statusbit nach links
RCR	Rotiere durch Übertrag-Statusbit nach rechts
REP	Wiederhole Zeichenkettenoperation bis Zähler CX gleich 0
REPE	Wiederhole Zeichenkettenoperation bis ungleich oder Zähler CX gleich 0
REPNE	Wiederhole Zeichenkettenoperation bis gleich oder Zähler CX gleich 0
REPNZ	Wiederhole Zeichenkettenoperation bis gleich 0 oder Zähler CX gleich 0
REPZ	Wiederhole Zeichenkettenoperation bis ungleich 0 oder Zähler CX gleich 0
RET	Kehre aus Unterprogramm zurück
ROL	Rotiere nach links
ROR	Rotiere nach rechts
SAHF	Kopiere AH-Register in die 8 Statusbits
SAL	Schiebe arithmetisch nach links
SAR	Schiebe arithmetisch nach rechts

Bezeichnung	Funktion
SBB	Subtrahiere mit Übertrag
SCAS	Durchsuche Zeichenkette
SCASB	Durchsuche Zeichenkette byteweise
SCASW	Durchsuche Zeichenkette wortweise
SEG	Definiere gültiges Segmentregister für nachfolgenden Befehl
SHL	Schiebe logisch nach links (entspricht SAL)
SHR	Schiebe logisch nach rechts
STC	Setze Übertrag-Statusbit
STD	Setze Richtung-Statusbit
STI	Setze Unterbrechung-Statusbit
STOS	Speichere Zeichenkette
STOSB	Speichere Zeichenkette byteweise
STOSW	Speichere Zeichenkette wortweise
SUB	Subtrahiere ohne Übertrag
TEST	Vergleiche über AND-Verknüpfung
WAIT	Warte auf Test-Signal
XCHG	Tausche Registerwerte (und Speicherinhalt) aus
XLAT	Adressiere Tabelle und lade Zeichen
XOR	Führe logische XOR-Verknüpfung aus

Tabelle 2. Die Funktion der Befehle des 8086/8088 gibt Aufschluß über die Leistungsfähigkeit des Prozessors

Die Befehle des 8086/8088 in alphabetischer Reihenfolge

Assemblercode	Maschinencode	Anweisung
AAA	37	Passe AX nach BCD-Addition an
AAD	D5, 0A	Passe AX vor BCD-Division an
AAM	D4, 0A	Passe AX nach BCD-Multiplikation an
AAS	3F	Passe AX nach ASCII-Subtraktion an
ADC AL/AX, data	0001010w, data	Addiere data und Übertrag-Statusbit zu AL/AX
ADC r/m, data	100000sw, mod 010 r/m [,addr/disp], data	Addiere data und Übertrag-Statusbit zu r/m
ADC r/m1, r/m2	000100dw, mod reg r/m [,addr/disp]	Addiere r/m2 und Übertrag-Statusbit zu r/m1
ADD AL/AX, data	0000010w, data	Addiere data zu AL/AX
ADD r/m, data	100000sw, mod 000 r/m [,addr/disp], data	Addiere data zu r/m
ADD r/m1, r/m2	000000dw, mod reg r/m [,addr/disp]	Addiere r/m2 zu r/m1
AND AL/AX, data	0010010w, data	Verknüpfe data mit AL/AX über die logische AND-Funktion
AND r/m, data	1000000w, mod 100 r/m [,addr/disp], data	Verknüpfe data mit r/m über die logische AND-Funktion
AND r/m1, r/m2	001000dw, mod reg r/m [,addr/disp]	Verknüpfe r/m2 mit r/m1 über die logische AND-Funktion
CALL address	9A, addr1, addr2	Rufe Unterprogramm ab addr1 (CS=addr2) auf
CALL disp16	E8, disp16	Rufe Unterprogramm ab disp16+naddr auf
CALL mem	FF, mod 011 r/m [,addr/disp]	Rufe Unterprogramm ab mem (CS=mem+2) auf
CALL r/m16	FF, mod 010 r/m [,addr/disp]	Rufe Unterprogramm ab r/m auf
CBW	98	Wandle Byte in AL in ein Wort für AX um
CLC	F8	Lösche Übertrag-Statusbit
CLD	FC	Lösche Richtung-Statusbit
CLI	FA	Lösche Unterbrechung-Statusbit
CMC	F5	Invertiere Übertrag-Statusbit
CMP AL/AX, data	0011110w, data	Vergleiche data mit AL/AX
CMP r/m, data	100000sw, mod 111 r/m [,addr/disp], data	Vergleiche data mit r/m
CMP r/m1, r/m2	001110dw, mod reg r/m [,addr/disp]	Vergleiche r/m2 mit r/m1
CMPS	1010011w	Vergleiche Zeichenkette ab SI mit Zeichenkette ab DI
CMPSB	A6	Vergleiche Zeichenkette ab SI byteweise mit Zeichenkette ab DI
CMPSW	A7	Vergleiche Zeichenkette ab SI wortweise mit Zeichenkette ab DI
CWD	99	Wandle Wort in AX in ein Doppelwort für DX (MSB) und AX (LSB) um
DAA	27	Passe AL nach dezimaler Addition an
DAS	2F	Passe AL nach dezimaler Subtraktion an
DEC reg16	01001reg16	Vermindere Register reg16 um 1
DEC r/m	1111111w, mod 001 r/m [,addr/disp]	Vermindere r/m um 1
DIV r/m	1111011w, mod 110 r/m [,addr/disp]	Dividiere DX:AX oder AX durch r/m (vorzeichenbehaftet)
ESC	11011xxx, mod xxx r/m [,addr/disp]	Übergebe r/m an externe Einheit
HLT	F4	Halte Prozessor an
IDIV r/m	1111011w, mod 111 r/m [,addr/disp]	Dividiere DX:AX oder AX durch r/m (vorzeichenlos)
IMUL r/m	1111011w, mod 101 r/m [,addr/disp]	Multipliziere AL/AX mit r/m (vorzeichenlos)
IN AL/AX, (DX)	1110110w	Lese Wert über Port (DX) in AL/AX
IN AL/AX, port	1110010w, port	Lies Wert über den angegebenen Port in AL/AX
INC reg16	01000reg16	Erhöhe Register reg16 um 1
INC r/m	1111111w, mod 000 r/m [,addr/disp]	Erhöhe r/m um 1
INT data8	CD, data8	Führe Software-Unterbrechung data8 aus
INT 3	CC	Führe Software-Unterbrechung 3 aus
INTO	CE	Führe Software-Unterbrechung 4 aus, wenn Überlauf (IP=Inhalt von 00010, CS=Inhalt von 00012)
IRET	CF	Kehre von Interrupt-Routine zurück, lies CS vom Stapel

Assemblercode	Maschinencode	Anweisung
JA disp8	77, disp8	Verzweige zu disp8+naddr, wenn darüber (C=0 und Z=0)
JAE disp8	73, disp8	Verzweige zu disp8+naddr, wenn darüber oder gleich (C=0)
JB disp8	72, disp8	Verzweige zu disp8+naddr, wenn darunter (C=1)
JBE disp8	76, disp8	Verzweige zu disp8+naddr, wenn darunter oder gleich (C=1 oder Z=1)
JCXZ disp8	E3, disp8	Verzweige zu disp8+naddr, wenn Inhalt von CX gleich 0
JE disp8	74, disp8	Verzweige zu disp8+naddr, wenn gleich (Z=1)
JG disp8	7F, disp8	Verzweige zu disp8+naddr, wenn größer (S=0 und Z=0)
JGE disp8	7D, disp8	Verzweige zu disp8+naddr, wenn größer oder gleich (O=S)
JL disp8	7C, disp8	Verzweige zu disp8+naddr, wenn kleiner (O< >S)
JLE disp8	7E, disp8	Verzweige zu disp8+naddr, wenn kleiner oder gleich (Z=1 oder O< >S)
JMP address	EA, addr1, addr2	Verzweige zu addr1 (CS=addr2)
JMP disp8	EB, disp8	Verzweige zu disp8+naddr
JMP disp16	E9, disp16	Verzweige zu disp16+naddr
JMP mem	FF, mod 101 r/m [,addr/disp]	Verzweige zu mem (CS=mem+2)
JMP r/m16	FF, mod 100 r/m [,addr/disp]	Verzweige zu r/m16+naddr
JNA disp8	76, disp8	Verzweige zu disp8+naddr, wenn nicht darüber (C=1 oder Z=1)
JNAE disp8	72, disp8	Verzweige zu disp8+naddr, wenn nicht darüber und nicht gleich (C=1)
JNB disp8	73, disp8	Verzweige zu disp8+naddr, wenn nicht darunter (C=0)
JNBE disp8	77, disp8	Verzweige zu disp8+naddr, wenn nicht darunter und nicht gleich (C=0 und Z=0)
JNE disp8	75, disp8	Verzweige zu disp8+naddr, wenn nicht gleich (Z=0)
JNG disp8	7E, disp8	Verzweige zu disp8+naddr, wenn nicht größer (Z=1 oder O< >S)
JNGE disp8	7C, disp8	Verzweige zu disp8+naddr, wenn nicht größer und nicht gleich (O< >S)
JNL disp8	7D, disp8	Verzweige zu disp8+naddr, wenn nicht kleiner (O=S)
JNLE disp8	7F, disp8	Verzweige zu disp8+naddr, wenn nicht kleiner und nicht gleich (S=0 und Z=0)
JNO disp8	71, disp8	Verzweige zu disp8+naddr, wenn kein Überlauf (O=0)
JNP disp8	7B, disp8	Verzweige zu disp8+naddr, wenn keine Parität (P=0)
JNS disp8	79, disp8	Verzweige zu disp8+naddr, wenn positiv (S=0)
JNZ disp8	75, disp8	Verzweige zu disp8+naddr, wenn nicht Null (Z=0)
JO disp8	70, disp8	Verzweige zu disp8+naddr, wenn Überlauf (O=1)
JP disp8	7A, disp8	Verzweige zu disp8+naddr, wenn Parität (P=1)
JPE disp8	7A, disp8	Verzweige zu disp8+naddr, wenn Parität gerade (P=1)
JPO disp8	7B, disp8	Verzweige zu disp8+naddr, wenn Parität ungerade (P=0)
JS disp8	78, disp8	Verzweige zu disp8+naddr, wenn negativ (S=1)
JZ disp8	74, disp8	Verzweige zu disp8+naddr, wenn 0 (Z=1)
LAHF	9F	Lade die unteren 8 Statusbits in AH
LDS reg16, mem	C5, mod reg16 r/m [,addr/disp]	Lade reg16 mit mem und DS mit mem+2
LEA reg16, mem	8D, mod reg16 r/m [,addr/disp]	Lade effektive Adresse von mem in reg16
LES reg16, mem	C4, mod reg16 r/m [,addr/disp]	Lade reg16 mit mem und ES mit mem+2
LOCK	F0	Aktiviere die LOCK-Signalleitung des Prozessors für den nachfolgenden Befehl
LODS	1010110w	Lade Zeichenkette ab SI in AL/AX
LODSB	AC	Lade Zeichenkette ab SI byteweise in AL
LODSW	AD	Lade Zeichenkette ab SI wortweise in AX
LOOP disp8	E2, disp8	Wiederhole Schleife bis CX=0 (CX wird jeweils um 1 dekrementiert)
LOOPE disp8	E1, disp8	Wiederhole Schleife bis CX=0 oder Z=0 (CX wird jeweils um 1 dekrementiert)
LOOPNE disp8	E0, disp8	Wiederhole Schleife bis CX=0 oder Z=1 (CX wird jeweils um 1 dekrementiert)
LOOPNZ disp8	E0, disp8	Wiederhole Schleife bis CX=0 oder Z=1 (CX wird jeweils um 1 dekrementiert)
LOOPZ disp8	E1, disp8	Wiederhole Schleife bis CX=0 oder Z=0 (CX wird jeweils um 1 dekrementiert)
MOV reg, data	1011wreg, data	Übertrage data in reg
MOV r/m, data	1100011w, mod 000 r/m [,addr/disp], data	Übertrage data in r/m
MOV AL/AX, addr	1010000w, mem	Übertrage mem in AL/AX
MOV addr, AL/AX	1010001w, mem	Übertrage AL/AX in mem
MOV r/m1, r/m2	100010dw, mod reg r/m [,addr/disp]	Übertrage r/m2 in r/m1
MOV sr, r/m16	8E, mod 0sr r/m [,addr/disp]	Übertrage r/m16 in sr
MOV r/m16, sr	8C, mod 0sr r/m [,addr/disp]	Übertrage sr in r/m16
MOVS	1010010w	Übertrage Zeichenkette ab SI nach DI
MOVESB	A4	Übertrage Zeichenkette byteweise ab SI nach DI
MOVESW	A5	Übertrage Zeichenkette wortweise ab SI nach DI
MUL r/m	1111011w, mod 100 r/m [,addr/disp]	Multipliziere AL/AX mit r/m (vorzeichenbehaftet)
NEG r/m	1111011w, mod 011 r/m [,addr/disp]	Negiere Wert von r/m (bilde 2er-Komplement)
NOP	90	Führe keine Operation aus
NOT r/m	1111011w, mod 010 r/m [,addr/disp]	Invertiere Wert von r/m (bilde 1er-Komplement)
OR AL/AX, data	0000110w, data	Verknüpfe data mit AL/AX über die logische OR-Funktion
OR r/m, data	1000000w, mod 001 r/m [,addr/disp], data	Verknüpfe data mit r/m über die logische OR-Funktion
OR r/m1, r/m2	000010dw, mod reg r/m [,addr/disp]	Verknüpfe r/m2 mit r/m1 über die logische OR-Funktion
OUT (DX), AL/AX	1110111w	Gebe AL/AX über Port (DX) aus
OUT port, AL/AX	1110011w, port	Gebe AL/AX über Port port aus
POP reg16	01011reg16	Lese in Register reg16 ein Wort vom Stapel
POP sr	000sr111	Lese in Segmentregister sr ein Wort vom Stapel
POP r/m16	8F, mod 000 r/m [,addr/disp]	Lese in r/m16 ein Wort vom Stapel
POPF	9D	Lese Statusbits vom Stapel
PUSH reg16	01010reg16	Lege Register reg16 auf den Stapel
USH sr	000sr110	Lege Segmentregister sr auf den Stapel
PUSH r/m16	FF, mod 110 r/m [,addr/disp]	Lege r/m16 auf den Stapel
PUSHF	9C	Lege Statusbits auf den Stapel
RCL r/m [,CL]	110100cw, mod 010 r/m [,addr/disp]	Rotiere r/m um 1 [CL] nach links und durch das Übertrag-Statusbit

Assemblercode	Maschinencode	Anweisung
RCR r/m [,CL]	110100cw, mod 011 r/m [,addr/disp]	Rotiere r/m um 1 [CL] nach rechts und durch das Übertrag-Statusbit
REP	F2 oder F3	Wiederhole Zeichenkettenoperation bis CX=0 (CX wird jeweils um 1 dekrementiert)
REPE	F3 (nur vor CMPS/SCAS)	Wiederhole Zeichenkettenoperation bis CX=0 oder Z=0 (CX wird jeweils um 1 dekrementiert)
REPNE	F2 (nur vor CMPS/SCAS)	Wiederhole Zeichenkettenoperation bis CX=0 oder Z=1 (CX wird jeweils um 1 dekrementiert)
REPNZ	F2 (nur vor CMPS/SCAS)	Wiederhole Zeichenkettenoperation bis CX=0 oder Z=1 (CX wird jeweils um 1 dekrementiert)
REPZ	F3 (nur vor CMPS/SCAS)	Wiederhole Zeichenkettenoperation bis CX=0 oder Z=0 (CX wird jeweils um 1 dekrementiert)
RET	CB	Kehre vom Unterprogramm zurück, lese CS vom Stapel
RET	C3	Kehre vom Unterprogramm zurück, lasse CS unverändert
RET disp16	CA, disp16	Kehre vom Unterprogramm zurück, lese CS vom Stapel, addiere disp16 zu SP
RET disp16	C2, disp16	Kehre vom Unterprogramm zurück, lasse CS unverändert, addiere disp16 zu SP
ROL r/m [,CL]	110100cw, mod 000 r/m [,addr/disp]	Rotiere r/m um 1 [CL] nach links
ROR r/m [,CL]	110100cw, mod 001 r/m [,addr/disp]	Rotiere r/m um 1 [CL] nach rechts
SAHF	9E	Lade AH in die unteren 8 Statusbits
SAL r/m [,CL]	110100cw, mod 100 r/m [,addr/disp]	Schiebe r/m um 1 [CL] arithmetisch nach links (entspricht SHL)
SAR r/m [,CL]	110100cw, mod 111 r/m [,addr/disp]	Schiebe r/m um 1 [CL] arithmetisch nach rechts
SBB AL/AX, data	0001110w, data	Subtrahiere data und das Übertrag-Statusbit von AL/AX
SBB r/m, data	100000sw, mod 011 r/m [,addr/disp], data	Subtrahiere data und das Übertrag-Statusbit von AL/AX
SBB r/m1, r/m2	000110dw, mod reg r/m [,addr/disp]	Subtrahiere r/m2 und das Übertrag-Statusbit von r/m1
SCAS	1010111w	Durchsuche Zeichenkette ab DI nach AL/AX
SCASB	AE	Durchsuche Zeichenkette ab DI byteweise nach AL
SCASW	AF	Durchsuche Zeichenkette ab DI wortweise nach AX
SEG	001sr110	Definiere sr als gültiges Segmentregister für den nächsten Befehl
SHL r/m [,CL]	110100cw, mod 100 r/m [,addr/disp]	Schiebe r/m um 1 [CL] logisch nach links (entspricht SAL)
SHR r/m [,CL]	110100cw, mod 101 r/m [,addr/disp]	Schiebe r/m um 1 [CL] logisch nach rechts
STC	F9	Setze Übertrag-Statusbit
STD	FD	Setze Richtung-Statusbit
STI	FB	Setze Unterbrechung-Statusbit
STOS	1010101w	Speichere AL/AX in Zeichenkette ab DI
STOSB	AA	Speichere AL byteweise in Zeichenkette ab DI
STOSW	AB	Speichere AX wortweise in Zeichenkette ab DI
SUB AL/AX, data	0010110w, data	Subtrahiere data von AL/AX
SUB r/m, data	100000sw, mod 101 r/m [,addr/disp], data	Subtrahiere data von r/m
SUB r/m1, r/m2	001010dw, mod reg r/m [,addr/disp]	Subtrahiere r/m2 von r/m1
TEST AL/AX, data	1010100w, data	Vergleiche data mit AL/AX über die AND-Funktion
TEST r/m, data	1111011w, mod 000 r/m [,addr/disp], data	Vergleiche data mit r/m über die AND-Funktion
TEST r/m1, r/m2	1000010w, mod reg r/m [,addr/disp]	Vergleiche r/m2 mit r/m1 über die AND-Funktion
WAIT	9B	Warte auf aktives Signal am Testeingang des Prozessors
XCHG AX, reg16	10010 reg16	Tausche Register reg16 gegen AX aus
XCHG r/m1, r/m2	1000011w, mod reg r/m [,addr/disp]	Tausche r/m2 gegen r/m1 aus
XLAT	D7	Übertrage den Inhalt der durch AL+BX+DS adressierten Speicherzelle in AL
XOR AL/AX, data	0011010w, data	Verknüpfe data mit AL/AX über die logische XOR-Funktion
XOR r/m, data	1000000w, mod 110 r/m [,addr/disp], data	Verknüpfe data mit r/m über die logische XOR-Funktion
XOR r/m1, r/m2	001100dw, mod reg r/m [,addr/disp]	Verknüpfe r/m2 mit r/m1 über die logische XOR-Funktion

Tabelle 3. Die Assemblerbefehle des 8086/8088 in ihrer Variation mit Maschinencode und exakter Bedeutung

Die Befehle des 8086/8088-Prozessors in numerischer Reihenfolge

Maschinencode	Befehl	1. Parameter	2. Parameter	Bemerkung
00, mod reg r/m	ADD	r/m8,	reg8	
01, mod reg r/m	ADD	r/m16,	reg16	
02, mod reg r/m	ADD	reg8,	r/m8	
03, mod reg r/m	ADD	reg16,	r/m16	
04, data8	ADD	AL,	data8	
05, data16	ADD	AX,	data16	
06	PUSH	ES		
07	POP	ES		
08, mod reg r/m	OR	r/m8,	reg8	
09, mod reg r/m	OR	r/m16,	reg16	
0A, mod reg r/m	OR	reg8,	r/m8	
0B, mod reg r/m	OR	reg16,	r/m16	
0C, data8	OR	AL,	data8	
0D, data16	OR	AX,	data16	
0E	PUSH	CS		
0F	POP	CS		
10, mod reg r/m	ADC	r/m8,	reg8	
11, mod reg r/m	ADC	r/m16,	reg16	
12, mod reg r/m	ADC	reg8,	r/m8	
13, mod reg r/m	ADC	reg16,	r/m16	
14, data8	ADC	AL,	data8	
15, data16	ADC	AX,	data16	
16	PUSH	SS		
17	POP	SS		
18, mod reg r/m	SBB	r/m8,	reg8	
19, mod reg r/m	SBB	r/m16,	reg16	
1A, mod reg r/m	SBB	reg8,	r/m8	
1B, mod reg r/m	SBB	reg16,	r/m16	
1C, data8	SBB	AL,	data8	
1D, data16	SBB	AX,	data16	
1E	PUSH	DS		
1F	POP	DS		

Maschinencode	Befehl	1. Parameter	2. Parameter	Bemerkung
20, mod reg r/m	AND	r/m8,	reg8	
21, mod reg r/m	AND	r/m16,	reg16	
22, mod reg r/m	AND	reg8,	r/m8	
23, mod reg r/m	AND	reg16,	r/m16	
24, data 8	AND	AL,	data8	
25, data16	AND	AX,	data16	
26	SEG	ES		
27	DAA			
28, mod reg r/m	SUB	r/m8,	reg8	
29, mod reg r/m	SUB	r/m16,	reg16	
2A, mod reg r/m	SUB	reg8,	r/m8	
2B, mod reg r/m	SUB	reg16,	r/m16	
2C, data8	SUB	AL,	data8	
2D, data16	SUB	AX,	data16	
2E	SEG	CS		
2F	DAS			
30, mod reg r/m	XOR	r/m8,	reg8	
31, mod reg r/m	XOR	r/m16,	reg16	
32, mod reg r/m	XOR	reg8,	r/m8	
33, mod reg r/m	XOR	reg16,	r/m16	
34, data8	XOR	AL,	data8	
35, data16	XOR	AX,	data16	
36	SEG	SS		
37	AAA			
38, mod reg r/m	CMP	r/m8,	reg8	
39, mod reg r/m	CMP	r/m16,	reg16	
3A, mod reg r/m	CMP	reg8,	r/m8	
3B, mod reg r/m	CMP	reg16,	r/m16	
3C, data8	CMP	AL,	data8	
3D, data16	CMP	AX,	data16	
3E	SEG	DS		
3F	AAS			

40	INC	AX	48	DEC	AX
41	INC	CX	49	DEC	CX
42	INC	DX	4A	DEC	DX
43	INC	BX	4B	DEC	BX
44	INC	SP	4C	DEC	SP
45	INC	BP	4D	DEC	BP
46	INC	SI	4E	DEC	SI
47	INC	DI	4F	DEC	DI
50	PUSH	AX	58	POP	AX
51	PUSH	CX	59	POP	CX
52	PUSH	DX	5A	POP	DX
53	PUSH	BX	5B	POP	BX
54	PUSH	SP	5C	POP	SP
55	PUSH	BP	5D	POP	BP
56	PUSH	SI	5E	POP	SI
57	PUSH	DI	5F	POP	DI

60 bis 6F ohne Verwendung

70, disp8	JO	disp8	
71, disp8	JNO	disp8	
72, disp8	JB/JNAE	disp8	
73, disp8	JAE/JNB	disp8	
74, disp8	JE/JZ	disp8	
75, disp8	JNE/JNZ	disp8	
76, disp8	JBE/JNA	disp8	
77, disp8	JA/JNBE	disp8	
78, disp8	JS	disp8	
79, disp8	JNS	disp8	
7A, disp8	JP/JPE	disp8	
7B, disp8	JPO/JNP	disp8	
7C, disp8	JL/JNGE	disp8	
7D, disp8	JGE/JNL	disp8	
7E, disp8	JLE/JNG	disp8	
7F, disp8	JG/JNLE	disp8	

Maschinencode	Befehl	1. Parameter	2. Parameter	Bemerkung
80, mod 000 r/m	ADD	r/m8,	data8	
80, mod 001 r/m	OR	r/m8,	data8	
80, mod 010 r/m	ADC	r/m8,	data8	
80, mod 011 r/m	SBB	r/m8,	data8	
80, mod 100 r/m	AND	r/m8,	data8	
80, mod 101 r/m	SUB	r/m8,	data8	
80, mod 110 r/m	XOR	r/m8,	data8	
80, mod 111 r/m	CMP	r/m8,	data8	
81, mod 000 r/m	ADD	r/m16,	data16	
81, mod 001 r/m	OR	r/m16,	data16	
81, mod 010 r/m	ADC	r/m16,	data16	
81, mod 011 r/m	SBB	r/m16,	data16	
81, mod 100 r/m	AND	r/m16,	data16	
81, mod 101 r/m	SUB	r/m16,	data16	
81, mod 110 r/m	XOR	r/m16,	data16	
81, mod 111 r/m	CMP	r/m16,	data16	
82, mod 000 r/m	ADD	r/m8,	data8	
82, mod 001 r/m	OR	r/m8,	data8	
82, mod 010 r/m	ADC	r/m8,	data8	
82, mod 011 r/m	SBB	r/m8,	data8	
82, mod 100 r/m	AND	r/m8,	data8	
82, mod 101 r/m	SUB	r/m8,	data8	
82, mod 110 r/m	XOR	r/m8,	data8	
82, mod 111 r/m	CMP	r/m8,	data8	
83, mod 000 r/m	ADD	r/m16,	data8	
83, mod 001 r/m	OR	r/m16,	data8	
83, mod 010 r/m	ADC	r/m16,	data8	
83, mod 011 r/m	SBB	r/m16,	data8	
83, mod 100 r/m	AND	r/m16,	data8	
83, mod 101 r/m	SUB	r/m16,	data8	
83, mod 110 r/m	XOR	r/m16,	data8	
83, mod 111 r/m	CMP	r/m16,	data8	
84, mod reg r/m	TEST	r/m8,	r/m8	
85, mod reg r/m	TEST	r/m16,	r/m16	
86, mod reg r/m	XCHG	r/m8,	r/m8	
87, mod reg r/m	XCHG	r/m16,	r/m16	
88, mod reg r/m	MOV	r/m8,	reg8	
89, mod reg r/m	MOV	r/m16,	reg16	
8A, mod reg r/m	MOV	reg8,	r/m8	
8B, mod reg r/m	MOV	reg16,	r/m16	
8C, mod 0sr r/m	MOV	r/m16,	sr	
8D, mod reg16 r/m	LEA	reg16,	mem	
8E, mod 0sr r/m	MOV	sr,	r/m16	
8F, mod 000 r/m	POP	r/m16		
90	NOP			
91	XCHG	AX,	CX	
92	XCHG	AX,	DX	
93	XCHG	AX,	BX	
94	XCHG	AX,	SP	
95	XCHG	AX,	BP	
96	XCHG	AX,	SI	
97	XCHG	AX,	DI	
98	CBW			
99	CWD			
9A, addr1, addr2	CALL	address		IP=addr1, CS=addr2
9B	WAIT			
9C	PUSHF			
9D	POPF			
9E	SAHF			
9F	LAHF			
A0, addr	MOV	AL,	mem	
A1, addr	MOV	AX,	mem	
A2, addr	MOV	mem,	AL	
A3, addr	MOV	mem,	AX	
A4	MOVSB			
A5	MOVSW			
A6	CMPSB			
A7	CMPSW			

Maschinencode	Befehl	1. Parameter	2. Param.	Bemerkung
A8, data8	TEST AL,	data8		
A9, data16	TEST AX,	data16		
AA	STOSB			
AB	STOSW			
AC	LODSB			
AD	LODSW			
AE	SCASB			
AF	SCASW			
B0, data8	MOV	AL,	data8	
B1, data8	MOV	CL,	data8	
B2, data8	MOV	DL,	data8	
B3, data8	MOV	BL,	data8	
B4, data8	MOV	AH,	data8	
B5, data8	MOV	CH,	data8	
B6, data8	MOV	DH,	data8	
B7, data8	MOV	BH,	data8	
B8, data16	MOV	AX,	data16	
B9, data16	MOV	CX,	data16	
BA, data16	MOV	DX,	data16	
BB, data16	MOV	BX,	data16	
BC, data16	MOV	SP,	data16	
BD, data16	MOV	BP,	data16	
BE, data16	MOV	SI,	data16	
BF, data16	MOV	DI,	data16	
C0	–			
C1	–			
C2, disp16	RET	disp16		CS unverändert (Intersegment)
C3	RET			CS unverändert (Intersegment)
C4, mod reg16 r/m	LES	reg16,	mem	
C5, mod reg16 r/m	LDS	reg16,	mem	
C6, mod 000 r/m	MOV	r/m8,	data8	
C7, mod 000 r/m	MOV	r/m16,	data16	
C8	–			
C9	–			
CA, disp16	RET	disp16		CS geändert (Intersegment)
CB	RET			CS geändert (Intersegment)
CC	INT	3		Abbruchpunkt (Breakpoint)
CD, data8	INT	data8		
CE	INTO			CS geändert (Intersegment)
CF	IRET			CS geändert (Intersegment)
D0, mod 000 r/m	ROL	r/m8		
D0, mod 001 r/m	ROR	r/m8		
D0, mod 010 r/m	RCL	r/m8		
D0, mod 011 r/m	RCR	r/m8		
D0, mod 100 r/m	SAL/SHL	r/m8		
D0, mod 101 r/m	SHR	r/m8		
D0, mod 110 r/m	–			
D0, mod 111 r/m	SAR	r/m8		
D1, mod 000 r/m	ROL	r/m16		
D1, mod 001 r/m	ROR	r/m16		
D1, mod 010 r/m	RCL	r/m16		
D1, mod 011 r/m	RCR	r/m16		
D1, mod 100 r/m	SAL/SHL	r/m16		
D1, mod 101 r/m	SHR	r/m16		
D1, mod 110 r/m	–			
D1, mod 111 r/m	SAR	r/m16		
D2, mod 000 r/m	ROL	r/m8,	CL	
D2, mod 001 r/m	ROR	r/m8,	CL	
D2, mod 010 r/m	RCL	r/m8,	CL	
D2, mod 011 r/m	RCR	r/m8,	CL	
D2, mod 100 r/m	SAL/SHL	r/m8,	CL	
D2, mod 101 r/m	SHR	r/m8,	CL	
D2, mod 110 r/m	–			
D2, mod 111 r/m	SAR	r/m8,	CL	

Maschinencode	Befehl	1. Parameter	2. Param.	Bemerkung
D3, mod 000 r/m	ROL	r/m16,	CL	
D3, mod 001 r/m	ROR	r/m16,	CL	
D3, mod 010 r/m	RCL	r/m16,	CL	
D3, mod 011 r/m	RCR	r/m16,	CL	
D3, mod 100 r/m	SAL/SHL	r/m16,	CL	
D3, mod 101 r/m	SHR	r/m16,	CL	
D3, mod 110 r/m	–			
D3, mod 111 r/m	SAR	r/m16,	CL	
D4, 0A	AAM			
D5, 0A	AAD			
D6	–			
D7	XLAT			
D8, mod xxx r/m	ESC			
D9, mod xxx r/m	ESC			
DA, mod xxx r/m	ESC			
DB, mod xxx r/m	ESC			
DC, mod xxx r/m	ESC			
DD, mod xxx r/m	ESC			
DE, mod xxx r/m	ESC			
DF, mod xxx r/m	ESC			
E0, disp8	LOOPNE/ LOOPNZ	disp8		
E1, disp8	LOOPE/ LOOPZ	disp8		
E2, disp8	LOOP	disp8		
E3, disp8	JCXZ	disp8		
E4, port	IN	AL,	port	
E5, port	IN	AX,	port	
E6, port	OUT	port,	AL	
E7, port	OUT	port,	AX	
E8, disp16	CALL	disp16		CS unverändert (Intersegment)
E9, disp16	JMP	disp16		CS unverändert (Intersegment)
EA, addr1, addr2	JMP	address		IP=addr1, CS=addr2
EB, disp8	JMP	disp8		CS unverändert (Intersegment)
EC	IN	AL,	(DX)	
ED	IN	AX,	(DX)	
EE	OUT	(DX),	AL	
EF	OUT	(DX),	AX	
F0	LOCK			
F1	–			
F2	REP/ REPNE/ REPNZ			
F3	REPE/ REPZ			
F4	HLT			
F5	CMC			
F6, mod 000 r/m	TEST	r/m8,	data8	
F6, mod 001 r/m	–			
F6, mod 010 r/m	NOT	r/m8		
F6, mod 011 r/m	NEG	r/m8		
F6, mod 100 r/m	MUL	r/m8		
F6, mod 101 r/m	IMUL	r/m8		
F6, mod 110 r/m	DIV	r/m8		
F6, mod 111 r/m	IDIV	r/m8		
F7, mod 000 r/m	TEST	r/m16,	data16	
F7, mod 001 r/m	–			
F7, mod 010 r/m	NOT	r/m16		
F7, mod 011 r/m	NEG	r/m16		
F7, mod 100 r/m	MUL	r/m16		
F7, mod 101 r/m	IMUL	r/m16		
F7, mod 110 r/m	DIV	r/m16		
F7, mod 111 r/m	IDIV	r/m16		

Maschinencode	Befehl	1. Parameter	2. Param.	Bemerkung
F8	CLC			
F9	STC			
FA	CLI			
FB	STI			
FC	CLD			
FD	STD			
FE, mod 000 r/m	INC	r/m8		
FE, mod 001 r/m	DEC	r/m8		
FE, mod 010 r/m	–			
FE, mod 011 r/m	–			
FE, mod 100 r/m	–			
FE, mod 101 r/m	–			
FE, mod 110 r/m	–			
FE, mod 111 r/m	–			
FF, mod 000 r/m	INC	r/m16		
FF, mod 001 r/m	DEC	r/m16		
FF, mod 010 r/m	CALL	r/m16		CS unverändert (Intersegment)
FF, mod 011 r/m	CALL	mem		CS geändert (Intersegment)
FF, mod 100 r/m	JMP	r/m16		CS unverändert (Intersegment)
,FF, mod 101 r/m	JMP	mem		CS geändert (Intersegment)
FF, mod 110 r/m	PUSH	r/m16		
FF, mod 111 r/m	–			

Tabelle 4. Alle Assemblerbefehle numerisch nach Maschinencode sortiert

mod.	1. Byte	2. Byte	3. Byte	4. Byte
00	Befehlscode	mod reg mem	–	–
00	Befehlscode	mod reg 110	addr (LSB)	addr (MSB)
01	Befehlscode	mod reg mem	disp8	–
10	Befehlscode	mod reg mem	disp16 (LSB)	disp16 (MSB)
11	Befehlscode	mod reg reg	–	–

Tabelle 5. Der Parameter »mod« bestimmt in einem Assemblerbefehl, wie dieser Befehl aufgebaut ist

reg	w=0	w=1
000	AL	AX
001	CL	CX
010	DL	DX
011	BL	BX
100	AH	SP
101	CH	BP
110	DH	SI
111	BH	DI

Tabelle 6. Der Parameter »reg« wählt in Zusammenarbeit mit »w« eines von 16 Registern aus

Code	Register
000	AX
001	CX
010	DX
011	BX
100	SP
101	BP
110	SI
111	DI

Tabelle 6a. Der Parameter »reg16« definiert eines der acht 16-Bit-Register

Code	Segmentregister
00	ES
01	CS
10	SS
11	DS

Tabelle 7. Der Parameter »sr« bestimmt für einen Assemblerbefehl ein Segmentregister

Spezielle Regeln für die Tabellen 3 und 4:

Der optionale Ausdruck [addr/disp] bezieht sich auf die durch »mem« angegebene Adresse einer Speicherzelle. Bei Befehlen in der Art »OP-CODE r/m1, r/m2« muß mindestens einer der beiden Operanden in einem Register stehen. Die Angaben »darüber« und »darunter« in den bedingten Sprungbefehlen beziehen sich auf den Vergleich vorzeichenloser Werte. Die Angaben »größer« und »kleiner« in den bedingten Sprungbefehlen beziehen sich auf den Vergleich von Werten mit Vorzeichen.

r/m	mod=00	mode=01	mod=10	mod=11 w=0	mod=11 w=1
000	BX+SI	BX+SI+disp8	BX+SI+disp16	AL	AX
001	BX+DI	BX+DI+disp8	BX+DI+disp16	CL	CX
010	BP+SI	BP+SI+disp8	BP+SI+disp16	DL	DX
011	BP+DI	BP+DI+disp8	BP+DI+disp16	BL	BX
100	SI	SI+disp8	SI+disp16	AH	SP
101	DI	DI+disp8	DI+disp16	CH	BP
110	addr	BP+disp8	BP+disp16	DH	SI
111	BX	BX+disp8	BX+disp16	BH	DI

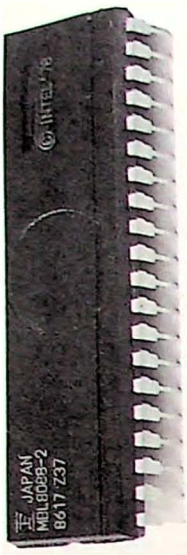
Tabelle 9. Mit den beiden Parametern »r/m« und »mod« läßt sich die Adressierungsart für einen Assemblerbefehl festlegen

Register/Statusbit	Funktion
AH	Obere 8 Bit des Akkumulators
AL	Untere 8 Bit des Akkumulators
AX	Akkumulator (accumulator)
BP	Basiszeiger (base pointer)
BH	Obere 8 Bit des Basisregisters
BL	Untere 8 Bit des Basisregisters
BX	Basisregister (base register)
C	Übertrag-Statusbit (carry flag)
CH	Obere 8 Bit des Programmregisters
CL	Untere 8 Bit des Programmregisters
CS	Programmsegment (code segment)
CX	Zählerregister (count register)
DH	Obere 8 Bit des Datenregisters
DI	Zielregister (destination index)
DL	Untere 8 Bit des Datenregisters
DS	Datensegment (data segment)
DX	Datenregister (data register)
ES	Extrasegment (extra segment)
FLAGS	Statusbits (status flags)
IP	Befehlszeiger (instruction pointer)
O	Überlauf-Statusbit (overflow flag)
P	Parität-Statusbit (parity flag)
S	Vorzeichen-Statusbit (sign flag)
SI	Quellregister (source index)
SP	Stapelzeiger (stack pointer)
SS	Stapelsegment (stack segment)
Z	Null-Statusbit (zero flag)

Tabelle 10. Die Bedeutung der Register und Statusbits, die in Tabelle 3 und 4 verwendet werden

Parameter	Bedeutung
addr	16-Bit-Adresse einer Speicherzelle (niederwertiges Byte vor höherwertigem Byte)
address	20-Bit-Adresse einer Speicherzelle, die sich aus einer 16-Bit-Adresse und einer nachfolgenden 16-Bit-Segmentadresse (für CS) zusammensetzt
AL/AX	w=0: Register AL angesprochen w=1: Register AX angesprochen
c	c=0: Operand um eine Bitposition verschieben c=1: Operand um die in CL angegebene Anzahl von Positionen verschieben
d	d=0: Ergebnis steht im durch mod und r/m adressierten Operanden d=1: Ergebnis steht im durch den Code reg bezeichneten Register
data	8- oder 16-Bit-Datenwert
data8	8-Bit-Datenwert
disp	(«displacement«) 8- oder 16-Bit-Adresse, die zu der dem aktuellen Befehl folgenden Adresse naddr addiert wird
disp8	8-Bit-Adresse, die zu der dem aktuellen Befehl folgenden Adresse naddr addiert wird
disp16	16-Bit-Adresse, die zu der dem aktuellen Befehl folgenden Adresse naddr addiert wird
mod	Code für eine Adressierungsart (siehe Tabelle)
mem	Inhalt der durch addr oder disp adressierten Speicherzelle(n)
naddr	Nächste gültige Adresse hinter dem aktuellen Befehl
port	8-Bit-Adresse eines Ports
reg	Code für ein Register (siehe Tabelle)
reg16	Code für ein 16-Bit-Register (siehe Tabelle)
r/m	Register oder Speicherzelleninhalt
r/m8	8-Bit-Register oder Inhalt einer Speicherzelle
r/m16	16-Bit-Register oder Inhalt zweier aufeinanderfolgenden Speicherzellen
sr	Code für ein Segmentregister (siehe Tabelle)
sw	sw=00: data ist ein 8-Bit-Operand sw=01: data ist ein 16-Bit-Operand sw=10: data ist ein 8-Bit-Operand sw=11: data ist ein 8-Bit-Operand, der auf 16 Bit erweitert werden muß
w	w=0: Bytebefehl w=1: Wortbefehl
x	Wahlweise gesetztes oder gelöschtes Bit

Tabelle 11. Diese Parameter werden ebenfalls in Tabelle 3 und 4 verwendet



Die zweite Stufe: der 80286

Bei der Entwicklung des 80286 dachten die Ingenieure von Intel vor allem an das Betriebssystemkonzept der Zukunft: Multitasking. Aber nicht alle 23 neuen Maschinenbefehle beziehen sich ausschließlich darauf, auch das Arbeiten mit den Ein-/Ausgabe-Ports wird mit dem 80286 wesentlich erleichtert.

Schon früh hatten es die Hard- und Software-Entwickler erkannt: Neuentwicklungen sind immer dann besonders erfolgreich, wenn Sie zu älteren Versionen kompatibel sind. Daher legte man bei der Konzeption des 80286-Prozessors Wert auf Kompatibilität zum Vorgänger 8086/8088. Alle Programme, die auf einem PC laufen, funktionieren daher auch ohne Einschränkungen auf einem AT. Trotzdem verdienen die neuen Befehle und Eigenschaften des 80286 Beachtung, auch wenn Sie unter MS-DOS normalerweise nicht verwendet werden. Der 16-Bit-Prozessor bekam folgende Neuerungen spendiert: eine zweite, als »Protected Mode« bezeichnete Betriebsart, die für den Ablauf von Multitasking- und Multiuser-Betriebssystemen konzipiert ist, vier zusätzliche Adreßleitungen A20 bis A23 zur direkten Adressierung eines 16 MByte (statt 1 MByte) großen Speicherbereichs, eine eigene MMU (»Memory Management Unit«), welche die für den Multitasking- und Multiuser-Betrieb vorgesehene neue Art der Speicherverwaltung mit verschiedenen Speicherschutzmechanismen unterstützt, ein Maschinenstatusregister (»MSW«) für prozessorrelevante Informationen und interne Betriebszustände, zwei neue Statuskennzeichen für die aktuelle Privilegstufe (»IOPL«) und Verschachtelung (»N«) und 23 neue Assemblerbefehle, um die erweiterten Fähigkeiten des 80286 auszunutzen (Tabelle 1). Neu hinzugekommen sind auch vier versteckte Register, und die Segmentregister sind um einen verdeckten Anhang erweitert. Doch dazu später mehr.

Wenn Sie diese Aufzählung etwas genauer untersuchen, werden Sie feststellen, daß mit dem 80286 gegenüber dem 8086/8088-Prozessor ein völlig neues Konzept verwirklicht wurde. Der 80286 ist bereits von der Hardware-Architektur voll auf Multitasking- und Multiuser-Betrieb ausgelegt, auch wenn im normalen PC-Alltag diese Eigenschaften meist brachliegen.

So wurde beispielsweise der allgemeine Registersatz des 80286 gegenüber dem 8086/8088-Prozessor um drei Bit im Statusbitregister und um ein vier Bit breites Maschinenstatusregister MSW (»Machine Status Word«) für prozessorrelevante Informationen und interne Betriebszustände erweitert. Bild 1 zeigt den kompletten Registersatz des 80286, Tabelle 2 den Aufbau des Statusbitregisters und Tabelle 3 die Bits des Maschinenstatusregisters.

Zusätzlich verfügt der 80286, wie bereits der 8086, über einen 6-Byte-Befehlspuffer (»Instruction Fetch Queue«) für die Aufnahme von Maschinencodebefehlen. Mit diesem Befehlspuffer kann der 80286 vorausschauend arbeiten und bereits einen neuen Befehl decodieren, während der vorherige noch ausgeführt wird.

Und selbstverständlich bearbeitet der 80286 viele Befehle wesentlich schneller als der 8086/8088. Bei diesen Kommandos läuft nun nicht mehr ein eigenes Microcodeprogramm im Prozessor ab, die entsprechenden Schaltanweisungen sind vielmehr als direktes »Hardware-Rechenwerk« implementiert.

Der 80286-Prozessor besitzt ein zusätzliches Statusregister und einen Befehlspuffer

Während sich Registersatz und Adressierungsarten des 80286 vom 8086/8088-Prozessor kaum beziehungsweise gar nicht unterscheiden, hat sich beim Befehlssatz schon mehr geändert. Die für den normalen PC-Programmierer sicherlich wichtigste Neuerung stellen die Befehle zur Ein- und Ausgabe von Zeichenketten (»ins« und »outs«), zum Anlegen und Löschen von Speicherstrukturen für Hochsprachen (»enter« und »leave«) sowie zum Laden und Speichern des Maschinenstatusregisters (»lmsw« und »smsw«) dar. Auch lassen sich beim 80286 alle Prozessorregister mit einem einzigen Befehl sichern und zurückholen (»popa« und »pusha«). Der Großteil der übrigen neuen Befehle zählt zu den sogenannten privilegierten Befehlen, welche die besonderen Fähigkeiten des 80286 im Protected Mode unterstützen und unter MS-DOS nicht benötigt werden.

Die größten Unterschiede des 80286 zum 8086/8088-Prozessor offenbaren

Die Zusatzbefehle des 80286-Prozessors

Befehl	Bedeutung
arpl	RPL-Feld eines Selektors korrigieren (RPL: »Requested Privilege Level«)
bound	Register auf Grenzwert testen
clts	TS-Bit im MSW auf 0 setzen
enter	Speicherstruktur auf dem Stapel anlegen
ins	Zeichenkette über Portadresse lesen
lar	Steuerbyte eines Selektors laden
leave	Speicherstruktur auf dem Stapel freigeben
lgdt	GDT-Register beschreiben (GDT: »Global Descriptor Table«)
lidt	IDT-Register beschreiben (IDT: »Interrupt Descriptor Table«)
lldt	LDT-Register beschreiben (LDT: »Local Descriptor Table«)
lmsw	MSW-Register beschreiben (MSW: »Machine Status Word«)
lsl	Segmentgrenze in Register laden
ltr	TR-Register beschreiben
outs	Zeichenkette über Portadresse ausgeben
popa	Alle Register vom Stapel holen
pusha	Alle Register auf den Stapel legen
sgdt	GDT-Register auslesen
sidt	IDT-Register auslesen
sldt	LDT-Register auslesen
smsw	MSW-Register auslesen
str	TR-Register auslesen
verr	Segment auf Lese-Freigabe überprüfen
verw	Segment auf Schreib-Freigabe überprüfen

Tabelle 1. Gegenüber dem 8086/8088 bekam der 80286-Prozessor 23 neue Befehle spendiert

sich im Protected Mode, in dem die vielen Eigenschaften zum Tragen kommen, die den 80286 für Multitasking- und Multiuser-Betriebssysteme so interessant machen. Die wichtigste dieser Eigenschaften ist die Speicherverwaltung über die interne MMU.

Im »Real Mode« ist der 80286 nicht von seinem kleinen Bruder, dem 8086/8088, zu unterscheiden

Die erste Betriebsart, in der sich der 80286 nach dem Einschalten oder einem Reset befindet, ist mit der (einzigen) Betriebsart des 8086/8088-Prozessor identisch. Sie wird beim 80286 mit »Real Mode« (»wahrer Modus«) bezeichnet und emuliert haargenau die Eigenschaften des 8086/8088, daß heißt, in diesem Modus sind die Adreßleitungen A20 bis A23 unbenutzt, und die Segmentregister werden auf die gleiche Art und Weise wie beim 8086/8088 verwendet.

Jedes Segmentregister bestimmt durch seinen Inhalt den Beginn eines Speichersegments, und die Adresse in dem zugehörigen Register bildet den Offset für dieses Segment. Das Programmsegment CS (Code Segment) ist für den Befehlszeiger IP, das Datensegment DS (Data Segment) für das Quellregister SI und das Zielregister DI, das Stapelsegment SS (Stack Segment) für den Stapelzeiger SP und den Basiszeiger BP sowie das Extrasegment ES (Extra Segment) für das Zielregister DI bei Zeichenkettenoperationen zuständig. Der 80286 bildet im Real Mode aus Segment- und Offset-Adresse die effektive Speicheradresse, indem er zu dem Wert des Registers, das die Offset-Adresse enthält, den mit 16 multiplizierten Wert des zugehörigen Segmentregisters addiert. So entsteht aus einer Segment- und einer Offset-Adresse ein 32-Bit-Zeiger auf eine bestimmte Stelle im Speicher.

Das Statusbitregister

Bit	Funktion
0	Übertrag-Statusbit C (Carry Flag)
1	unbenutzt
2	Parität-Statusbit P (Parity Flag)
3	unbenutzt
4	Hilfsübertrag-Statusbit A (Auxiliary Carry Flag)
5	unbenutzt
6	Null-Statusbit Z (Zero Flag)
7	Vorzeichen-Statusbit S (Sign Flag)
8	Einzelschritt-Statusbit T (Trace Flag)
9	Interrupt-Statusbit I (Interrupt Flag)
10	Richtung-Statusbit D (Direction Flag)
11	Überlauf-Statusbit O (Overflow Flag)
12 und 13	Privilegstufe IOPL (Input/Output Privilege Level)
14	Verschachtelung-Statusbit N (Nested Task Flag)
15	unbenutzt

Tabelle 2. Das Statusregister eines 80286

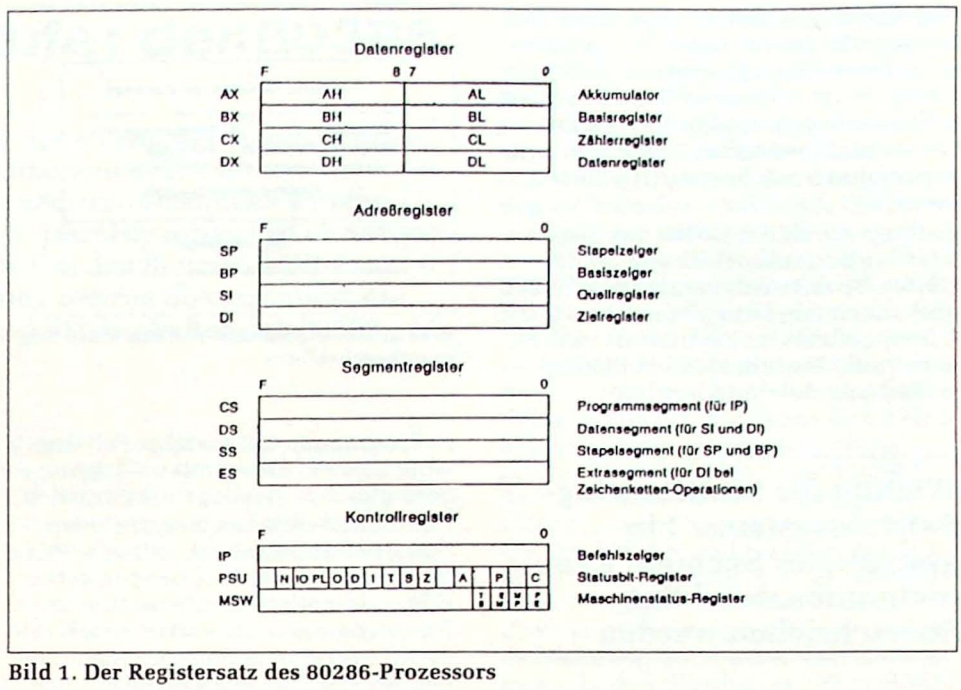


Bild 1. Der Registersatz des 80286-Prozessors

Was dem 8086/8088 seine Segmentadressierung, das sind dem 80286 seine Beschreibertabellen

Die zweite Betriebsart des 80286 wird als »Protected Mode« (»geschützter Modus«) bezeichnet. Im Protected Mode funktioniert das Verfahren der Segmentadressierung komplizierter, es ist aber auch leistungsfähiger. In diesem Modus repräsentiert der Inhalt eines Segmentregisters nämlich einen Zeiger auf eine globale oder lokale Beschreibertabelle im Arbeitsspeicher des Computers. Diese Beschreibertabellen enthalten wiederum die Adressen von globalen oder lokalen Segmentbeschreibern. Ein Segmentbeschreiber beinhaltet Informationen über Position und Länge eines globalen oder lokalen Speichersegments, über die Art des Speicherinhalts (Programmcode oder Daten), über die Schreib- und Leseberechtigung und die Privilegstufe. Bild 2 zeigt den allgemeinen Aufbau eines Segmentbeschreibers.

Für die Verwaltung der Beschreibertabellen sind die versteckten Register des 80286 zuständig:

- das GDT-Register, das einen Zeiger auf die globale Beschreibertabelle enthält,
- das LDT-Register, das einen Zeiger auf die lokale Beschreibertabelle enthält,
- das IDT-Register, das einen Zeiger auf die Interrupt-Beschreibertabelle enthält, die wiederum die Gates (»Tore«) für den Zugriff auf die Interrupt-Serviceroutinen für die 256 Software-Interrupts enthält
- und das Taskregister TR, das einen Zeiger auf die Beschreibertabelle enthält, welche die Segmentbeschreiber der Taskstate-Segmente enthält.

Der 80286 unterscheidet grundsätzlich zwischen zwei Typen von Segmentbeschreibern: die Datensegmentbeschreiber für Segmente, die ausschließlich Daten enthalten, und die Codesegmentbeschreiber für Segmente, die lauffähige Programme enthalten. Jeder Segmentbeschreiber besteht aus acht Byte. Den Start bildet ein 16-Bit-Wert, der die Länge des Segments angibt, darauf folgt eine 24-Bit-Basisadresse, welche die Start- oder Endadresse des Segments innerhalb des Speichers festlegt. Das folgende Byte ist ein Steuerbyte, das unter anderem Angaben über die Privilegstufe und über die Schreib- oder Leseberechtigung für das Segment enthält. Den Abschluß bilden 16 Bit, die aus Gründen der Aufwärtskompatibilität zum 32-Bit-Prozessor 80386 auf den Wert 0 gesetzt sein müssen.

Jedes der vier Segmentregister im 80286 verfügt über einen verdeckten Anhang von 6 Byte, in dem jeweils

Segmentlänge, Basisadresse und das Steuerbyte des aktuellen Segmentbeschreibers abgelegt werden. Dadurch ist der Segmentbeschreiber bei Bedarf sofort verfügbar und muß vom Prozessor nicht jedesmal über einen zeitraubenden Speicherzugriff geladen werden.

Bedingt durch die Größe des Segmentlängenparameters von 16 Bit dürfen Speichersegmente beim 80286 eine maximale Länge von $2^{16} = 64$ KByte aufweisen. Zumindest in diesem Punkt besteht also kein Unterschied zum Real Mode.

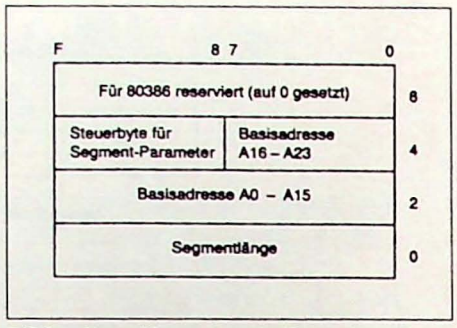


Bild 2. Der allgemeine Aufbau eines Segmentbeschreibers

Wichtig für Multitasking-Betriebssysteme: Ein geschütztes Segment kann nicht unbeabsichtigt überschrieben werden

Die Angabe der Segmentlänge im Beschreiber verhindert, daß Schreib- oder Lesezugriffe versehentlich außerhalb des Segments aufsetzen.

Über die 24-Bit-Basisadresse läßt sich jedes Segment innerhalb eines Speicherbereichs von 16 MByte frei positionieren.

Das Steuerbyte mit seinen Angaben über Privilegstufe sowie Schreib- und Leseberechtigung erlaubt es, bestimmte Speichersegmente mit einem Schutz zu versehen, um Lese- oder Schreibzugriffe auf Daten oder Programme zu verhindern, so daß beispielsweise wichtige Programme nicht von jedermann überschrieben und vertrauliche Daten von unberechtigten Anwendern nicht gelesen werden können.

Das Konzept der Privilegstufen erlaubt das Verteilen von Prioritäten an Anwender und Programme. Die aktuelle Privilegstufe wird ständig durch das Statuskennzeichen »iopl« des Statusbitregisters angezeigt. Insgesamt sind vier Privilegstufen möglich, Stufe 0 hat die höchste und Stufe 3 die niedrigste Wertigkeit. Ein Programm kann jeweils nur auf andere Programme mit gleicher Privilegstufe und auf Daten mit niedriger oder gleicher Privilegstufe zugreifen.

Dadurch, daß jedem Segment über den Beschreiber eine bestimmte Privilegstufe zugeordnet werden kann, läßt sich erreichen, daß bestimmte Programme nur auf vorher festgelegte Daten und Programme zugreifen dürfen. So kann beispielsweise der Systemprogrammierer für andere Anwender ganz einfach Zugriffe auf das Betriebssystem verhindern, indem er den Speichersegmenten mit den Betriebssystemroutinen die Privilegstufe 0 und allen anderen Programmen und Daten eine höhere Privilegstufe zuweist. Dieses Verfahren ist auch als Supervisor/User-Konzept bekannt.

Neben den Segmentbeschreibern benötigt der 80286 zur Verwaltung eines Multitasking- oder Multiuser-Betriebssystems noch die Systembeschreiber und die Gate-Beschreiber. Die Systembeschreiber sind wie Segmentbeschreiber aufgebaut, zeigen jedoch auf Segmente, die systemspezifische Informationen enthalten. Dieses sind lokale Beschreibertabellen oder Taskstate-Segmente, die Informationen über den Zustand eines Task enthalten und deshalb besonders schutzbedürftig sind.

In diesem Zusammenhang ist das Taskregister TR wichtig, das einen Zeiger auf die Beschreibertabelle enthält, die wiederum die Segmentbeschreiber der einzelnen Taskstate-Segmente enthält. Die Gate-Beschreiber sind zwar wie die Segment- und Systembeschreiber ebenfalls acht Byte lang, unterscheiden sich jedoch im Aufbau deutlich von diesen.

Über die Gate-Beschreiber ist es einem Programm möglich, das Segment oder die Privilegstufe zu wechseln. Dadurch kann auch ein wenig privilegiertes Programm in eine höhere Privilegstufe aufsteigen.

Der 80286 ist mit seinem Multitasking- und Multiuser-freundlichen Hardware-Konzept in der Lage, auch größere Computersysteme effektiv zu verwalten. Zwar wird der 80286-Prozessor im AT zur Zeit meist nur im Real Mode betrieben, weil MS-DOS dies aus Kompatibilitätsgründen fordert, so daß das Leistungspotential des 80286-Prozessors größtenteils ungenutzt bleibt, doch das Betriebssystem OS/2, das den Protected Mode benutzt, zeigt, wozu der 80286 fähig ist. (ma/tr)

Das Maschinenstatusregister

Bit	Funktion
0	Protected-Mode-Statusbit PE (Protected Mode Enable), zeigt den Betrieb im Protected Mode an
1	Arithmetikprozessor-Statusbit MP (Monitor Processor Extension), zeigt das Vorhandensein eines Arithmetikprozessors an
2	Emulation-Statusbit EM (Emulate Processor Extension), zeigt die Emulation eines Arithmetikprozessors an
3	Taskwechsel-Statusbit TS (Task Switch), zeigt einen Taskwechsel an

Tabelle 3. Das Maschinenstatusregister enthält nur vier Bit

80286: Prozessor ohne Zukunft?

Die Verkaufspolitik von Intel deutet darauf hin, die Produktion des 80286 stark zu drosseln oder sogar ganz einzustellen. Bei der Vorstellung des 80486 im Mai '89 sagte ein Intel-Manager wörtlich: »Das Ende des 80286 hat begonnen«. Der abgespeckten SX-Version des 80386-Prozessors wird eine große Zukunft vorhergesagt (siehe auch DOS International, Ausgabe 6'89, Seite 40). Müssen jetzt die Besitzer von »kleinen« ATs fürchten, in Zukunft von Intel keine Hard- und Software-Unterstützung mehr zu erhalten?

Die Realität sieht anders aus, selbst wenn Intel die Produktion des 80286 ganz einstellen sollte: 25-MHz-Versionen des 80286 sind inzwischen keine Seltenheit mehr und auch zu einem erschwinglichen Preis erhältlich. Die Firmen AMD und Harris zum Beispiel stützen sich derzeit ganz auf die Produktion dieses schnellen Prozessors. Und die Verarbeitungsgeschwindigkeit eines mit 16 MHz getakteten ATs liegt immerhin etwa 10 Prozent über der eines 16-MHz-386er mit einem 80386SX.

Für denjenigen Kunden, der einen sehr schnellen IBM-kompatiblen Rechner möchte, ist und bleibt der 80286 vor allem auch preislich sehr attraktiv. Wer allerdings Wert auf Multitasking-Umgebungen wie zum Beispiel Windows 286 oder Xenix legt, für den stellt ein Rechner mit einem 80386SX die bessere Wahl dar. Der 80286 wird jedoch erst dann völlig abgelöst sein, wenn 80386-Rechner bei gleicher Ausstattung und Verarbeitungsgeschwindigkeit nur unwesentlich mehr kosten als ein AT.

Die dritte Stufe: der 80386

Die wichtigste Eigenschaft des 80386, die diesen Prozessor gegenüber dem 8086/8088 und 80286 auszeichnet, ist die Fähigkeit, gleichzeitig mehrere 8086/8088-Prozessoren emulieren zu können. Darüber hinaus ist er natürlich voll abwärtskompatibel zu den 8- und 16-Bit-Prozessoren von Intel. Der folgende Beitrag dokumentiert die neuen Befehle, Register und Eigenschaften des 80386.

Wie sein kleinerer Bruder, der 80286, so kennt auch der 80386 zwei verschiedene Betriebsarten. Dies ist zum einen der sofort nach dem Einschalten aktive »Real Adress Mode«, oder kurz »Real Mode«, und zum anderen der »Protected Virtual Adress Mode«, auch »Protected Mode« genannt.

Im Real Mode verhält sich der 80386, vor allem in bezug auf die Speicheradressierung, wie ein 8086/8088, nur ist er eben wesentlich schneller. Die Geschwindigkeitssteigerung erreichten die Entwickler dadurch, daß viele Befehle durch die Implementierung als fest verdrahtete Logikschaltungen anstelle von Microcode nun in weniger Taktzyklen ausgeführt werden. Ein im Real Mode laufender 80386 hat einen maximal adressierbaren Speicherraum von 1 Megabyte. Über Bit 0 des MSW-Registers (»Machine Status Word«) schalten Sie jederzeit in den Protected Mode und wieder zurück in den Real Mode. Der Inhalt des MSW-Registers entspricht dem des 80286-Prozessors.

Im Protected Mode beträgt eine Segmentlänge nicht mehr 64 KByte, sondern 4 GByte (1 Gigabyte = 1024 MByte), und der insgesamt adressierbare Speicher nicht 1 MByte (8086/8088) oder 16 MByte (80286), sondern ebenfalls 4 GByte, virtuell sogar 64 TByte (1 Terabyte = 1024 GByte). Virtuell deshalb, weil dieser Wert das Produkt aus dem beim 80386 maximal erlaubten 16384 Segmenten zu je 4 GByte Länge darstellt. Doch ist dieser Wert nur der maximale logische Adreßraum, der physikalische wird durch den »nur« 32 Bit breiten Adreßbus des 80386 weiterhin auf 4 GByte beschränkt.

Nur im Protected Mode stehen die erweiterten Fähigkeiten des 80386 zur Verfügung, wie zum Beispiel der automatische Wechsel zwischen laufenden Programmen (»Tasks«), das Sperren von Zugriffen auf Speicherbereiche oder Portadressen der Peripherie und so weiter.

Ebenfalls neu beim 80386 ist der »Virtual 8086 Mode«, kurz als Virtual Mode bezeichnet. Dieser Modus ist eigentlich ein Untermodus des Protected Mode und verwaltet gleichzeitig bis zu 256 Real-Mode-Umgebungen, so daß der 80386 entsprechend mehrere Programme, die im Real Mode laufen, gleichzeitig bearbeiten kann. Das Statusbit »VM« (Bild 1) zeigt an, ob sich der zur Zeit aktive Task im Virtual Mode befindet. Das bekannteste Programm, das den Virtual Modus nutzt, ist der MS-DOS-Aufsatz Windows 386. Mit Windows 386 ist es auf einem 386er-PC möglich, mehrere MS-DOS-Programme parallel auszuführen.

Die 32 internen Register des 80386 enthalten aus Kompatibilitätsgründen zunächst einmal den kompletten Registersatz des 8086/8088. Die rechte Hälfte der auf 32 Bit Länge erweiterten Register EAX, EBX, ECX, EDX, EDI, ESI, EBP und ESP (Bild 2) stellen die vollwertigen Register AX, BX, CX, DX, DI, SI, BP und SP dar, wie sie im 8086/8088 und im 80286 verwendet werden. Wenn Sie also zum Beispiel mit AX die ersten 16 Bit des EAX-Registers ansprechen, dann bleiben die Bit 16 bis 31 unbenutzt und werden auch nicht verändert.

Über die sechs Segmentregister CS, DS, ES, FS, GS und SS (Bild 3) können Sie im Proteced Mode also insgesamt 24 GByte adressieren, ohne überhaupt einen Segmentwechsel durchführen zu müssen. Interessant ist auch, daß die eben genannten Register bei einem Task-Wechsel automatisch ausgetauscht werden.

Im Real oder Virtual Mode des Prozessors werden nur die letzten 16 Bit des CS-Registers verwendet und zeigen wie beim 8086/8088 auf ein 64 KByte langes Segment. Der Inhalt des IP-Registers dient dann als Offset innerhalb dieses Segments.

Der 80386 verwaltet im Protected Mode Speichersegmente von 4 GByte Länge

Der Befehlszeiger EIP bildet im Protected Mode einen 32-Bit-Offset, der über das aktive Codesegment CS auf das nächste auszuführende Kommando deutet. Ein Segment ist daher 4 GByte groß. Im Gegensatz zum 8086/8088 beträgt der fest vorgegebene Mindestabstand zwischen zwei Segmenten aber nicht 16 Byte, sondern 4 KByte.

Eine weitere Besonderheit des 80386 sind die acht Zusatzregister DR0 bis DR7. Die ersten vier dieser jeweils 16 Bit langen Debug-Register (DR) dienen zur Aufnahme einer Speicheradresse (Offset). Der Prozessor selbst überprüft während des Programmlaufs ständig, ob eine dieser Adressen erreicht wurde und erzeugt bei Bedarf eine Ausnahmebedingung. Der Vorteil liegt darin, daß der Programmierer Breakpoints auch im ROM des Computers setzen kann – mit einem Break-Kommando, das normalerweise an der entsprechenden Programmstelle einfach eingefügt wird, wäre dies nicht möglich. Ein anderer Vorteil der Debug-Register ist, daß sie nicht nur mit dem ak-

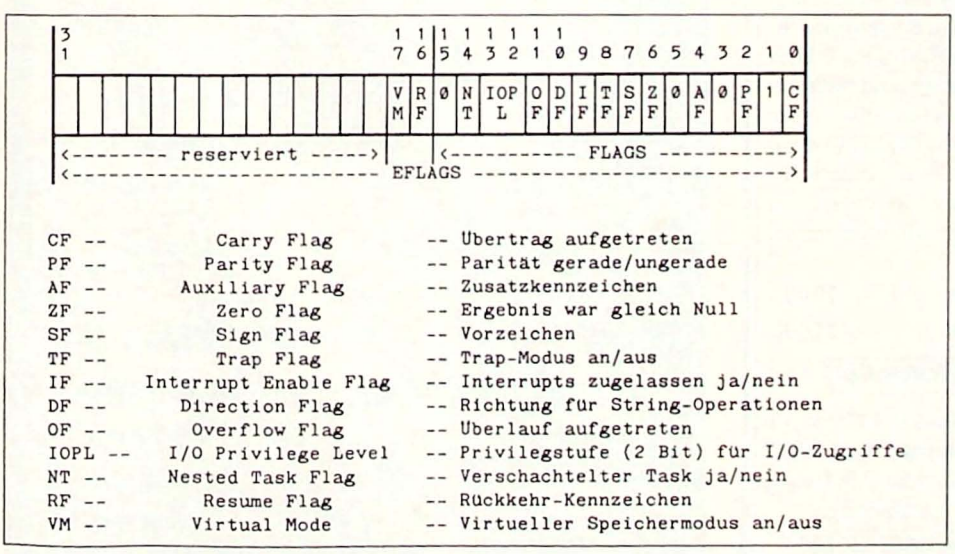


Bild 1. Das Statusbitregister des 80386

tuellen Stand des IP-Registers, sondern auch mit den anderen Segmentregistern verglichen werden. Dies erlaubt dem Programmierer zum Beispiel, den Schreibzugriff von Programmen auf eine bestimmte Speicherstelle innerhalb eines Datenbereichs zu überwachen.
Die Debug-Register DR6 und DR7 teilen dem Prozessor mit, welche der vier Adressen DR0 bis DR3 er überwachen soll und welche Bedingungen eine Unterbrechung auslösen. So kann der Programmierer beispielsweise definieren, daß bei einer bestimmten Adresse nur Programmcode stehen darf und Schreibzugriffe anderer Prozesse nicht erlaubt sind.

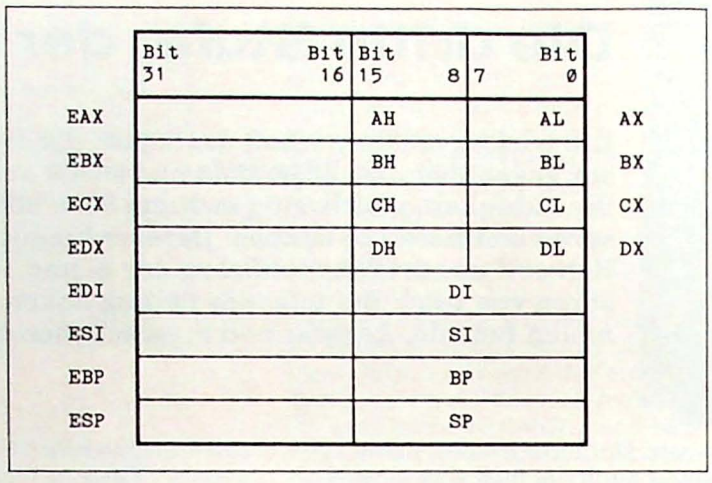


Bild 2. Die Register des 80386 sind gegenüber dem 8086 und 80286 auf 32 Bit erweitert

Als echter Multitasking-Prozessor ist der 80386 für die Verwaltung von mehreren Prozessen ausgelegt

Wie der 80286, so arbeitet auch der 80386 als Multitasking-Prozessor mit umfangreichen Schutzmechanismen für den Einsatz in Mehrprogramm-Umgebungen. Jeder Task bekommt eine von vier möglichen Privilegstufen zugeteilt und hat dann nur Zugriffsrechte auf Speicherbereiche, die von Tasks mit gleicher oder höherer Stufe belegt sind. Üblicherweise verwendet ein Betriebssystem diese Stufen so:

Stufe 0 – Betriebssystemkern (Kernel)

Stufe 1 – Betriebssystemumgebung

Stufe 2 – Anwendungsprogramme

Stufe 3 – Temporäre Daten

Unter einem Multitasking-Betriebssystem kommt der 80386 erst richtig zum Einsatz. Auf Anforderung führt er den Wechsel zwischen zwei Programmen fast vollautomatisch durch.

Er rettet zum Beispiel selbsttätig den kompletten Registersatz und lädt den neuen Inhalt. Dazu verwaltet der 80386 zu jedem Task eine Tabelle, die als »Task State Segment« bezeichnet wird. In dieser findet nicht nur der Registersatz Platz, auch Informationen über die noch offenen Dateien

Königsbergerstr.4
2000 Hamburg 71
SHC computer Fink
PEACOCK COMPUTER
Unsere Stärke ist Desktop Publishing

Pea Cock® Young Line

Der Young Line besticht durch sein junges, fröhliches Design.

Wir wünschen jeden diesen flotten AT Personal Computer im Büro.

AT 286 mit 10 MHz getaktet • 1 MB Hauptspeicher • 1 Diskettenlaufwerk 5,25" oder 3,5" 1,2 MB • Harddisk 20 MB 40 ms • 14" Monitor monochrom amber • 2 parallele und 1 serielle Schnittstelle • Bus-Mouse-Anschluß • erweiterte Tastatur (102) • MS- DOS 4.01® mit MS-Graphic Shell - Benutzeroberfläche • Mouse deutsche Handbücher. 3698,-

1 Jahr Garantie

• Pea Cock Flash (XT)
• Pea Cock AT 286 von 10-16 MHz
• Pea Cock AT 386 von 16-25 MHz

Die neuen von NEC®

Die 2 neuen Farb-Monitore Multisync 2A und Multisync 3D von NEC sind speziell für den anspruchsvollen Anwender entwickelt worden. So sind sie z.B. auch für den IBM® PS/2 User eine hochwertige und preislich interessante Alternative. Neue Techniken z.B. Digital Control beim 3D und das äußere Erscheinungsbild lassen sie zu Highlights am Monitorhimmel werden.

Multisync 2A 1799,-
Multisync 3D 2290,-

1 Jahr Garantie

Die Synthese aus Pea Cock AT Personal Computer und NEC Multisync 2A oder 3D ist *die* harmonische Verbindung.

Tel.040/6405614 + 6407509 • Fax.040/6401807

Microsoft
Schul- und Universitätshändler

Word	467,-	MS-DOS Lernprogramm	84,-
Multiplan	340,-	Quickbasic Compiler	150,-
Chart	361,-	Quick C-Compiler	150,-
Project	404,-	Mouse/PB (Bus\ser.\PS/2)	251,-
Excel	500,-	Mouse/WinPB (Bus\ser.\PS/2)	308,-
Works	247,-		
Clout	417,-		
Fortran	343,-		
Basic Comp.	343,-		
Cobol	587,-		
Pascal	309,-		
Windows 286	199,-		
Windows 386	247,-		

Alle Preise verstehen sich in DM inkl. MWSt., Die oben genannten Preise gelten nur für Schulen, Universitäten, Schüler und Studenten. Ihrer schriftlichen Bestellung muß ein Berechtigungsnachweis beigefügt sein (z.B. Studenten - Original der gültigen Immatrikulationsbescheinigung).

Fordern Sie unsere komplette Preisliste an!

datanorm computer systeme gmbh
3400 Göttingen
Schildweg 11
Tel. 05 51 / 48 59 38 - 39

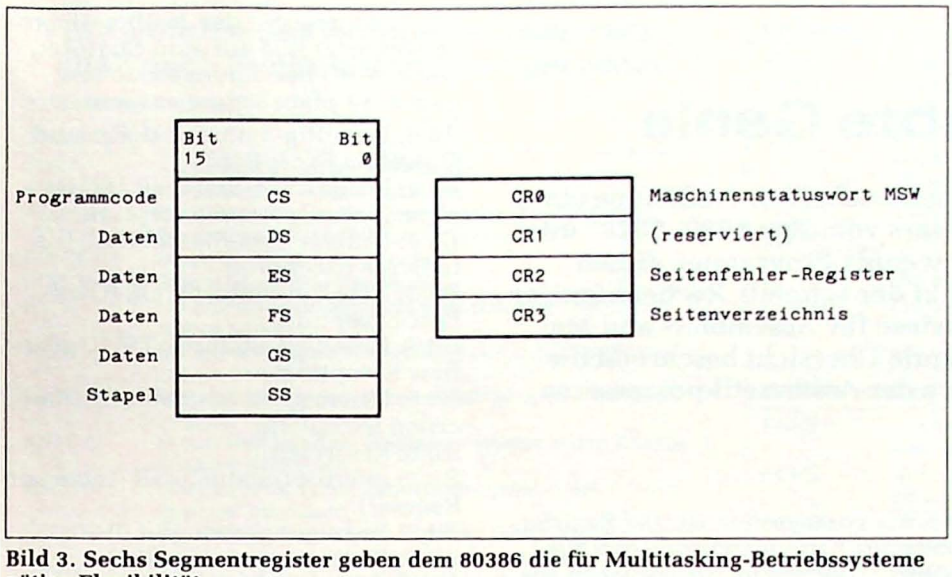


Bild 3. Sechs Segmentregister geben dem 80386 die für Multitasking-Betriebssysteme nötige Flexibilität

des jeweiligen Tasks, die Privilegien und ähnliches sind hier abgelegt. Der Zeiger auf das Task State Segment des gerade aktiven Tasks befindet sich im Task Register TR. Für den Systemprogrammierer ist der Wechsel zwischen zwei Tasks einfach: Er führt einen Jump-Befehl (»jump new task«) auf eine Adresse außerhalb des gültigen Bereichs auf den neu zu startenden Tasks aus, und alles weitere erledigt der Prozessor.

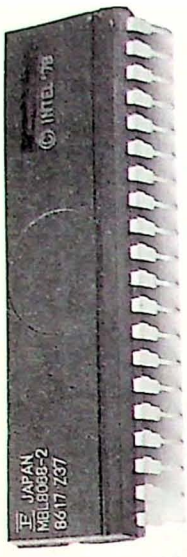
Wie bereits angedeutet, sind die meisten Zusatzfunktionen des 80386 unter MS-DOS nicht sinnvoll einzusetzen und nur für Systementwickler interessant. Der »normale« Assemblerprogrammierer profitiert vom erweiterten Befehlssatz, der alle Zusatzkommandos des 80286 mit einschließt, und natürlich von der wesentlich gesteigerten Ausführungsgeschwindigkeit. Die Tabelle enthält die gegenüber dem 80286 hinzugekommenen Instruktionen. Die genaue Erklärung der neuen Logikbefehle finden Sie im Referenzartikel »Logik« in dieser DOS-Extra-Ausgabe.

(tr)

Die neuen Befehle des 80386-Prozessors in alphabetischer Reihenfolge

Befehl	Englische Bedeutung	Deutsche Übersetzung
BSF	bit scan forward	Bit-Suchlauf vorwärts
BSR	bit scan reverse	Bit-Suchlauf rückwärts
BT	test bit	Teste Bits auf Gleichheit
BTC	test bit and complement	Teste Bits auf Gleichheit und invertiere sie
BTR	test bit and reset	Teste Bits auf Gleichheit und lösche sie
BTS	test bit and set	Teste Bits auf Gleichheit und setze sie
CLTS	clear task switched flag	Lösche das Prozeßwechselkennzeichen
JECXZ	jump on ECX zero	Verzweige, wenn das ECZ-Register Null ist
LFS	load pointer to FS	Lade einen Pointer in das FS-Register
LGS	load pointer to GS	Lade einen Pointer in das GS-Register
LSS	load pointer to SS	Lade einen Pointer in das SS-Register
MOVSX	move with sign extension	Verschiebe einen Wert und erweitere das Vorzeichen
MOVZX	move with zero extension	Verschiebe einen Wert mit einer Null-Erweiterung
SETB (SETNAE)	set byte on below (not above or equal)	Setze Byte, wenn darunter (nicht darüber oder gleich)
SETBE (SETNA)	set byte on below or equal (not above)	Setze Byte, wenn darunter oder gleich (nicht darüber)
SETE (SETZ)	set byte on equal (zero)	Setze Byte, wenn gleich (Null)
SETL (SETNGE)	set byte on less (not greater or equal)	Setze Byte, wenn weniger (nicht größer oder gleich)
SETLE (SETNG)	set byte on less or equal (not greater)	Setze Byte, wenn weniger oder gleich (nicht größer)
SETNB (SETAE)	set byte on not below (above or equal)	Setze Byte, wenn nicht darunter (darüber oder gleich)
SETNBE (SETA)	set byte on not below or equal (above)	Setze Byte, wenn nicht darunter oder gleich (darüber)
SETNE (SETNZ)	set byte on not equal (not zero)	Setze Byte, wenn nicht gleich (nicht Null)
SETNL (SETGE)	set byte on not less (greater or equal)	Setze Byte, wenn nicht weniger (größer oder gleich)
SETNLE (SETG)	set byte on not less or equal (greater)	Setze Byte, wenn nicht weniger oder gleich (größer)
SETNO	set byte on not overflow	Setze Byte, wenn kein Überlauf
SETNP (SETPO)	set byte on not parity (parity odd)	Setze Byte, wenn keine Parität (Parität ungerade)
SETNS	set byte on not sign	Setze Byte, wenn kein Vorzeichen
SETO	set byte on overflow	Setze Byte, wenn Überlauf
SETP (SETPE)	set byte on parity (parity even)	Setze Byte, wenn Parität (Parität gerade)
SETS	set byte on sign	Setze Byte, wenn Vorzeichen
SHLD	shift left double	Schiebe ein 32-Bit-Register bitweise nach links
SHRD	shift right double	Schiebe ein 32-Bit-Register bitweise nach rechts

Tabelle. Die neben dem vollen Befehlssatz der 8086/8088- und 80286-Prozessoren neuen Kommandos beim 80386



8087
Das ungeliebte Genie

Fast jeder PC hat einen freien Sockel zur Aufnahme eines mathematischen Coprozessors vom Typ 8087, 80287 oder 80387. Weil aber nur sehr wenige Programme diesen wirklich unterstützen, bleibt der schnelle Rechenkünstler auch weiterhin eine Spielwiese für Assembler- und Mathematik-Freaks. Die folgende Übersicht beschreibt die Register und Spezialbefehle der Arithmetikprozessoren von Intel.

Der Arithmetikprozessor 8087 wurde, ebenso wie seine größeren Brüder, von Intel zur Unterstützung des 8086/8088-Prozessors bei mathematischen Operationen entwickelt. So sind im Arithmetikprozessor 8087 viele mathematische Funktionen (Grundrechenarten, Wurzelberechnung, Winkelfunktionen, Logarithmen) fest integriert und werden in einem einzigen Programmschritt ausgeführt. Zudem rechnet der Arithmetikprozessor prinzipiell mit 80 Bit breiten numerischen Werten – höchste Genauigkeit und ein großer Wertebereich sind dadurch garantiert.

Die Programmierung des Arithmetikprozessors 8087 erfolgt vom 8086/8088-Prozessor aus über Befehlsfolgen, denen der Assemblerbefehl »ESC« vorangestellt ist. Der Registersatz des Arithmetikprozessors 8087 (Bild 1) besteht im wesentlichen aus einem Registerstapel von acht 80-Bit-Registern und einem Statusregister, das die Flags zur Kennzeichnung von Spezialfällen enthält.

Alle numerischen Werte legt der Arithmetikprozessor in den Registern des Registerstapels als temporäre Fließkommazahlen ab. Sollten die Werte noch nicht in diesem Format vorliegen, dann wandelt er sie automatisch um.

Tabelle 1 zeigt die numerischen Formate, die der Arithmetikprozessor verarbeiten kann, und Bild 2 zeigt, wie er numerische Werte in verschiedenen Formaten im Arbeitsspeicher des PC ablegt.

Das Statusregister des Arithmetikprozessors setzt sich aus acht Statusbit, einem 4-Bit-Bedingungscode und dem 3-Bit-Stapelzeiger zusammen:

Bit 0: Ungültig-Statusbit IE (Invalid Operation Exception)
Bit 1: Unmaskiert-Statusbit DE (Denormalized Operand Exception)
Bit 2: Nulldivision-Statusbit ZE (Zerodivide Exception)
Bit 3: Überlauf-Statusbit OE (Overflow Exception)
Bit 4: Unterlauf-Statusbit UE (Underflow Exception)
Bit 5: Genauigkeit-Statusbit PE (Precision Exception)
Bit 6: Reserviert
Bit 7: Interrupt-Statusbit IR (Interrupt Request)
Bit 8: Bedingungscode (Bit 0)
Bit 9: Bedingungscode (Bit 1)
Bit A: Bedingungscode (Bit 2)
Bit B: Stapelzeiger (Bit 0)
Bit C: Stapelzeiger (Bit 1)
Bit D: Stapelzeiger (Bit 2)
Bit E: Bedingungscode (Bit 3)
Bit F: Beschäftigt-Statusbit B (Busy)

Der Stapelzeiger in Bit B bis Dhex markiert mit seinem Inhalt jeweils das Register, das die Spitze des Registerstapels darstellt. Dieses Register trägt in den Assemblerbefehlen die Bezeichnung st(0), das darunterliegende Register die Bezeichnung st(7) und so weiter. Wird beispielsweise der Wert des Stapelzeigers um 1 erhöht, so wird das über st(0) liegende Register zum neuen Register st(0) und das alte st(0) zum neuen Register st(7).

Tabelle 3 zeigt die Assemblerbefehle des Arithmetikprozessors 8087 nach Funktionsgruppen geordnet und Tabelle 4 die genaue Bedeutung aller 8087-Assemblerbefehle. Die in den Tabellen 3 und 4 verwendeten Parameter finden Sie in Tabelle 2 erklärt.

(ma/tr)

Die Variablentypen des 8087

Typ	Länge	Wertebereich
word integer	2 Byte	-32768 bis +32767
short integer	4 Byte	$-2{,}14 \times 10^{+9}$ bis $+2{,}14 \times 10^{+9}$
long integer	8 Byte	$-9{,}22 \times 10^{+18}$ bis $+9{,}22 \times 10^{+18}$
packed decimal	10 Byte	-999999999999999999 bis +999999999999999999
short real	4 Byte	$+/- 8{,}43 \times 10^{+/-37}$ bis $+/- 3{,}37 \times 10^{+38}$
long real	8 Byte	$+/- 4{,}19 \times 10^{+/-307}$ bis $+/- 1{,}67 \times 10^{+308}$
temporary real	10 Byte	$+/- 3{,}4 \times 10^{-4932}$ bis $+/- 1{,}2 \times 10^{+4932}$

Tabelle 1. Diese sieben numerischen Formate verarbeitet der Arithmetikprozessor 8087

Die Parameter der für den Arithmetikprozessor 8087 gültigen Assemblerbefehle

Parameter	Bedeutung
mem	Speicherstelle im Arbeitsspeicher des PC
st	Kurzform für st(0)
st(i)	Register aus dem Registerstapel des Arithmetikprozessors (Wertebereich für i: 0 bis 7)
st(0)	Register aus dem Registerstapel des Arithmetikprozessors, auf das der Stapelzeiger gegenwärtig zeigt
st(1)	Register aus dem Registerstapel des Arithmetikprozessors, das dem Register st(0) unmittelbar folgt

Tabelle 2. Diese Parameter werden in den Tabellen 3 und 4 verwendet

Der Registersatz des Arithmetikprozessors 8087

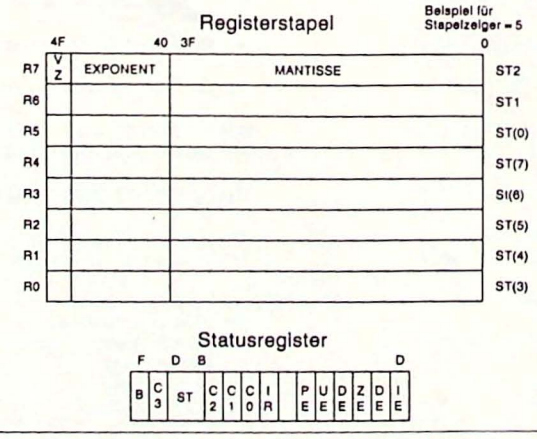


Bild 1. Der Registersatz des 8087 besteht hauptsächlich aus einem 80-Bit-Registerstapel und dem Statusregister

Die Befehle des Arithmetikprozessors 8087 nach Funktionsgruppen geordnet

Bezeichnung	Funktion
Steuerbefehle	
FINIT	Arithmetikprozessor initialisieren
FFREE	Register als leer kennzeichnen
FCLEX	Exception-Statusbits, Unterbrechungsanforderung-Statusbit und Busy-Statusbit löschen
FINCSTP	Stapelzeiger inkrementieren
FDECSTP	Stapelzeiger dekrementieren
FDISI	Unterbrechungsanforderungen durch Arithmetikprozessor sperren
FENI	Unterbrechungsanforderungen durch Arithmetikprozessor freigeben
FLDCW	Steuerwort aus dem Arbeitsspeicher in das Steuerwortregister lesen
FSTCW	Steuerwort aus dem Steuerwortregister in den Arbeitsspeicher schreiben
FSTSW	Statuswort aus dem Statuswortregister in den Arbeitsspeicher schreiben
FLDENV	Steuerwort, Statuswort, Tag-Feld, Befehlszeiger und Datenzeiger aus dem Arbeitsspeicher lesen
FSTENV	Steuerwort, Statuswort, Tag-Feld, Befehlszeiger und Datenzeiger in den Arbeitsspeicher schreiben
FRSTOR	Sämtliche Register des Arithmetikprozessors aus dem Arbeitsspeicher lesen
FSAVE	Sämtliche Register des Arithmetikprozessors in den Arbeitsspeicher schreiben
FNOP	Keine Operation ausführen
FWAIT	Auf Testsignal warten
Transferbefehle	
FLD	Fließkommazahl auf den Registerstapel legen
FST	Fließkommazahl vom Registerstapel lesen (Stapelzeiger bleibt unverändert)
FSTP	Fließkommazahl vom Registerstapel lesen
FXCH	Fließkommazahl des obersten Registers auf dem Stapel gegen Fließkommazahl eines anderen Registers austauschen
FILD	Ganzzahligen Wert aus dem Arbeitsspeicher auf den Registerstapel legen
FIST	Ganzzahligen Wert vom Registerstapel in den Arbeitsspeicher schreiben (Stapelzeiger bleibt unverändert)
FISTP	Ganzzahligen Wert vom Registerstapel in den Arbeitsspeicher schreiben
FBLD	Gepackte Dezimalzahl im BCD-Format aus dem Arbeitsspeicher auf den Registerstapel legen
FBSTP	Gepackte Dezimalzahl im BCD-Format vom Registerstapel in den Arbeitsspeicher schreiben
Addition	
FADD	Fließkommazahlen addieren
FADDP	Fließkommazahl im obersten Register des Stapels zur Fließkommazahl eines anderen Registers addieren
FIADD	Ganzzahligen Wert aus dem Arbeitsspeicher zur Fließkommazahl des obersten Registers auf dem Stapel addieren (Stapelzeiger bleibt unverändert)
Subtraktion	
FSUB	Fließkommazahlen subtrahieren
FSUBP	Fließkommazahl im obersten Register des Stapels von der Fließkommazahl eines anderen Registers subtrahieren
FISUB	Ganzzahligen Wert aus dem Arbeitsspeicher von der Fließkommazahl im obersten Register des Stapels subtrahieren (Stapelzeiger bleibt unverändert)
FSUBR	Fließkommazahlen umgekehrt subtrahieren
FSUBRP	Fließkommazahl eines Registers von der Fließkommazahl im obersten Register des Stapels subtrahieren
FISUBR	Fließkommazahl im obersten Register des Stapels von einem ganzzahligen Wert im Arbeitsspeicher subtrahieren (Stapelzeiger bleibt unverändert)
Multiplikation	
FMUL	Fließkommazahlen multiplizieren
FMULP	Fließkommazahl im obersten Register des Stapels mit Fließkommazahl eines anderen Registers multiplizieren
FIMUL	Ganzzahligen Wert aus dem Arbeitsspeicher mit Fließkommazahl im obersten Register des Stapels multiplizieren (Stapelzeiger bleibt unverändert)
Division	
FDIV	Fließkommazahlen dividieren
FDIVP	Fließkommazahl eines Registers durch Fließkommazahl im obersten Register des Stapels dividieren
FIDIV	Fließkommazahl im obersten Register des Stapels durch ganzzahligen Wert aus dem Arbeitsspeicher dividieren (Stapelzeiger bleibt unverändert)
FDIVR	Fließkommazahlen umgekehrt dividieren
FDIVRP	Fließkommazahl im obersten Register des Stapels durch Fließkommazahl eines anderen Registers dividieren
FIDIVR	Ganzzahligen Wert im Arbeitsspeicher durch Fließkommazahl im obersten Register des Stapels dividieren (Stapelzeiger bleibt unverändert)
Weitere arithmetische Befehle	
FABS	Absolutwert der Fließkommazahl im obersten Register des Stapels berechnen
FCHS	Vorzeichen der Fließkommazahl im obersten Register des Stapels komplementieren
FPREM	Rest bei der Division von zwei Fließkommazahlen bestimmen
FRNDINT	Fließkommazahl im obersten Register des Stapels in ganzzahligen Wert umwandeln
FSCALE	Fließkommazahl nach der Funktion $z = y * x * 2^x$ berechnen
FSQRT	Quadratwurzel aus der Fließkommazahl im obersten Register des Stapels berechnen
FXTRACT	Fließkommazahl im obersten Register des Stapels in Exponent und Mantisse aufteilen
Transzendente Befehle	
FPATAN	Arkustangens einer Fließkommazahl berechnen
FPTAN	Tangens einer Fließkommazahl berechnen
F2XM1	Fließkommazahl nach der Funktion $z = 2^x - 1$ berechnen
FYL2X	Fließkommazahl nach der Funktion $z = y * x * \log_2 * x$ berechnen
FYL2XP1	Fließkommazahl nach der Funktion $z = y * x * \log_2 * (x + 1)$ berechnen
Konstanten schreiben	
FLD1	Konstante 1 in das oberste Register des Stapels schreiben
FLDZ	Konstante 0 in das oberste Register des Stapels schreiben
FLDPI	Konstante Pi in das oberste Register des Stapels schreiben
FLDL2E	Konstante $\log_2 e$ in das oberste Register des Stapels schreiben
FLDL2T	Konstante $\log_2 10$ in das oberste Register des Stapels schreiben
FLDLG2	Konstante $\log_{10} 2$ in das oberste Register des Stapels schreiben
FLDLN2	Konstante $\log_e 2$ in das oberste Register des Stapels schreiben
Vergleichsbefehle	
FCOM	Fließkommazahl im obersten Register des Stapels mit einer anderen Fließkommazahl vergleichen (Stapelzeiger bleibt unverändert)
FCOMP	Fließkommazahl im obersten Register des Stapels mit einer anderen Fließkommazahl vergleichen
FCOMPP	Fließkommazahl im obersten Register des Stapels mit Fließkommazahl im nachfolgenden Register vergleichen, Stapelzeiger doppelt erhöhen
FICOM	Ganzzahligen Wert im Arbeitsspeicher mit Fließkommazahl im obersten Register des Stapels vergleichen (Stapelzeiger bleibt unverändert)
FICOMP	Ganzzahligen Wert im Arbeitsspeicher mit Fließkommazahl im obersten Register des Stapels vergleichen
FTST	Fließkommazahl im obersten Register des Stapels mit 0 vergleichen
FXAM	Fließkommazahl im obersten Register des Stapels untersuchen

Tabelle 3. Die Assemblerbefehle des Arithmetikprozessors 8087 nach Funktionsgruppen geordnet

Die für den Arithmetikprozessor 8087 gültigen Assemblerbefehle

Befehl	Parameter	Bedeutung
Steuerbefehle		
FINIT	–	Initialisiere Arithmetikprozessor
FFREE	st(i)	Kennzeichne Register st(i) als leer
FCLEX	–	Lösche Exception-Statusbits, Unterbrechungsanforderung-Statusbit und Busy-Statusbit
FINCSTP	–	Inkrementiere Stapelzeiger
FDECSTP	–	Dekrementiere Stapelzeiger
FDISI	–	Setze Unterbrechung-Statusbit (Unterbrechungsanforderungen durch Arithmetikprozessor gesperrt)
FENI	–	Lösche Unterbrechung-Statusbit (Unterbrechungsanforderungen durch Arithmetikprozessor freigegeben)
FLDCW	mem	Lies Steuerwort aus mem in Steuerwortregister
FSTCW	mem	Schreibe Steuerwort aus Steuerwortregister in mem
FSTSW	mem	Schreibe Statuswort aus Statuswortregister in mem
FLDENV	mem	Lies Steuerwort, Statuswort, Tag-Feld, Befehlszeiger und Datenzeiger ab mem (0Ehex Byte)
FSTENV	mem	Schreibe Steuerwort, Statuswort, Tag-Feld, Befehlszeiger und Datenzeiger ab mem (0Ehex Bytes)
FRSTOR	mem	Lies sämtliche Register des Arithmetikprozessors ab mem (5Ehex Byte)
FSAVE	mem	Schreibe sämtliche Register des Arithmetikprozessors ab mem (5Ehex Byte)
FNOP	–	Führe keine Operation aus
FWAIT	–	Warte auf aktives Signal am Testeingang
Transferbefehle		
FLD	st(i)	Dekrementiere Stapelzeiger, schreibe Wert aus st(i) in st(0)
FLD	mem	Dekrementiere Stapelzeiger, schreibe Fließkommazahl aus mem in st(0)
FST	st(i)	Schreibe Wert aus st(0) in st(i)
FST	mem	Schreibe Wert aus st(0) in mem
FSTP	st(i)	Schreibe Wert aus st(0) in st(i), inkrementiere Stapelzeiger
FSTP	mem	Schreibe Wert aus st(0) in mem, inkrementiere Stapelzeiger
FXCH	st(i)	Tausche Wert aus st(0) gegen Wert in st(i) aus
FXCH	–	Tausche Wert aus st(0) gegen Wert in st(1) aus
FILD	mem	Dekrementiere Stapelzeiger, schreibe ganzzahligen Wert aus mem in st(0)
FIST	mem	Schreibe Wert aus st(0) als ganzzahligen Wert in mem
FISTP	mem	Schreibe Wert aus st(0) als ganzzahligen Wert in mem, inkrementiere Stapelzeiger
FBLD	mem	Dekrementiere Stapelzeiger, schreibe Dezimalzahl im BCD-Format aus mem als Fließkommazahl in st(0)
FBSTP	mem	Schreibe Wert aus st(0) als gepackte Dezimalzahl im BCD-Format in mem, inkrementiere Stapelzeiger
Addition		
FADD	st,st(i)	Addiere Wert aus st(i) mit st(0), schreibe Ergebnis in st(0)
FADD	st(i),st	Addiere Wert aus st(0) mit st(i), schreibe Ergebnis in st(i)
FADD	mem	Addiere Fließkommazahl aus mem mit st(0), schreibe Ergebnis in st(0)
FADD	–	Addiere Wert aus st(0) mit st(1), schreibe Ergebnis in st(1), inkrementiere Stapelzeiger
FADDP	st(i),st	Addiere Wert aus st(0) mit st(i), schreibe Ergebnis in st(i), inkrementiere Stapelzeiger
FIADD	mem	Addiere ganzzahligen Wert aus mem mit st(0), schreibe Ergebnis in st(0)
Subtraktion		
FSUB	st,st(i)	Subtrahiere Wert aus st(i) von st(0), schreibe Ergebnis in st(0)
FSUB	st(i),st	Subtrahiere Wert aus st(0) von st(i), schreibe Ergebnis in st(i)
FSUB	mem	Subtrahiere Fließkommazahl aus mem von st(0), schreibe Ergebnis in st(0)
FSUB	–	Subtrahiere Wert aus st(0) von st(1), schreibe Ergebnis in st(1), inkrementiere Stapelzeiger
FSUBP	st(i),st	Subtrahiere Wert aus st(0) von st(i), schreibe Ergebnis in st(i), inkrementiere Stapelzeiger
FISUB	mem	Subtrahiere ganzzahligen Wert aus mem von st(0), schreibe Ergebnis in st(0)
FSUBR	st,st(i)	Subtrahiere Wert aus st(0) von st(i), schreibe Ergebnis in st(0)
FSUBR	st(i),st	Subtrahiere Wert aus st(i) von st(0), schreibe Ergebnis in st(i)
FSUBR	mem	Subtrahiere Wert aus st(0) von Fließkommazahl in mem, schreibe Ergebnis in st(0)
FSUBR	–	Subtrahiere Wert aus st(1) von st(0), schreibe Ergebnis in st(1), inkrementiere Stapelzeiger
FSUBRP	st(i),st	Subtrahiere Wert aus st(i) von st(0), schreibe Ergebnis in st(i), inkrementiere Stapelzeiger
FISUBR	mem	Subtrahiere Wert aus st(0) vom ganzzahligen Wert in mem, schreibe Ergebnis in st(0)
Multiplikation		
FMUL	st,st(i)	Multipliziere Wert aus st(i) mit st(0), schreibe Ergebnis in st(0)
FMUL	st(i),st	Multipliziere Wert aus st(0) mit st(i), schreibe Ergebnis in st(i)
FMUL	mem	Multipliziere Fließkommazahl aus mem mit st(0), schreibe Ergebnis in st(0)
FMUL	–	Multipliziere Wert aus st(0) mit st(1), schreibe Ergebnis in st(1), inkrementiere Stapelzeiger
FMULP	st(i),st	Multipliziere Wert aus st(0) mit st(i), schreibe Ergebnis in st(i), inkrementiere Stapelzeiger
FIMUL	mem	Multipliziere ganzzahligen Wert aus mem mit st(0), schreibe Ergebnis in st(0)
Division		
FDIV	st,st(i)	Dividiere Wert aus st(0) durch st(i), schreibe Ergebnis in st(0)
FDIV	st(i),st	Dividiere Wert aus st(i) durch st(0), schreibe Ergebnis in st(i)
FDIV	mem	Dividiere Wert aus st(0) durch Fließkommazahl in mem, schreibe Ergebnis in st(0)
FDIV	–	Dividiere Wert in st(1) durch st(0), schreibe Ergebnis in st(1), inkrementiere Stapelzeiger
FDIVP	st(i),st	Dividiere Wert aus st(i) durch st(0), schreibe Ergebnis in st(i), inkrementiere Stapelzeiger
FIDIV	mem	Dividiere Wert aus st(0) durch ganzzahligen Wert in mem, schreibe Ergebnis in st(0)
FDIVR	st,st(i)	Dividiere Wert aus st(i) durch st(0), schreibe Ergebnis in st(0)
FDIVR	st(i),st	Dividiere Wert aus st(0) durch st(i), schreibe Ergebnis in st(i)
FDIVR	mem	Dividiere Fließkommazahl aus mem durch st(0), schreibe Ergebnis in st(0)
FDIVR	–	Dividiere Wert aus st(0) durch st(1), schreibe Ergebnis in st(1), inkrementiere Stapelzeiger
FDIVRP	st(i),st	Dividiere Wert aus st(0) durch st(i), schreibe Ergebnis in st(i), inkrementiere Stapelzeiger
FIDIVR	mem	Dividiere ganzzahligen Wert aus mem durch st(0), schreibe Ergebnis in st(0)
Weitere arithmetische Befehle		
FABS	–	Berechne Absolutwert von Fließkommazahl aus st(0), schreibe Ergebnis in st(0)
FCHS	–	Komplementiere Vorzeichen der Fließkommazahl aus st(0), schreibe Ergebnis in st(0)
FPREM	–	Dividiere Wert aus st(0) durch st(1), schreibe Rest der Division in st(0)
FRNDINT	–	Wandle Fließkommazahl aus st(0) in ganzzahligen Wert, schreibe Ergebnis in st(0)
FSCALE	–	Berechne Funktion $y = st(0) * x * 2^{st(1)}$, schreibe Ergebnis in st(0)
FSQRT	–	Berechne Quadratwurzel von Wert aus st(0), schreibe Ergebnis in st(0)
FXTRACT	–	Teile Fließkommazahl aus st(0) in Exponent und Mantisse auf, schreibe Exponent in st(0), dekrementiere Stapelzeiger, schreibe Mantisse in st(0)

Transzendente Befehle		
FPATAN	–	Berechne Arkustangens aus dem Bruch in st(1) (Zähler) sowie st(0) (Nenner), inkrementiere Stapelzeiger, schreibe Ergebnis in st(0)
FPTAN	–	Berechne Tangens als Bruch aus dem Wert von st(0), schreibe Zähler in st(0), dekrementiere Stapelzeiger, schreibe Nenner in st(0)
F2XM1	–	Berechne Funktion $y = 2^{st(0)} - 1$, schreibe Ergebnis in st(0)
FYL2X	–	Berechne Funktion $y = st(1) * x * \log_2 st(0)$, inkrementiere Stapelzeiger, schreibe Ergebnis in st(0)
FYL2XP1	–	Berechne Funktion $y = st(1) * x * \log_2(st(0) + 1)$, inkrementiere Stapelzeiger, schreibe Ergebnis in st(0)
Konstanten schreiben		
FLD1	–	Dekrementiere Stapelzeiger, schreibe die Konstante 1 in st(0)
FLDZ	–	Dekrementiere Stapelzeiger, schreibe die Konstante 0 in st(0)
FLDPI	–	Dekrementiere Stapelzeiger, schreibe die Konstante Pi in st(0)
FLDL2E	–	Dekrementiere Stapelzeiger, schreibe die Konstante $\log_2 e$ in st(0)
FLDL2T	–	Dekrementiere Stapelzeiger, schreibe die Konstante $\log_2 10$ in st(0)
FLDLG2	–	Dekrementiere Stapelzeiger, schreibe die Konstante $\log_{10} 2$ in st(0)
FLDLN2	–	Dekrementiere Stapelzeiger, schreibe die Konstante $\log_e 2$ in st(0)

Vergleichsbefehle		
FCOM	st(i)	Vergleiche Wert aus st(i) mit st(0), setze entsprechend die Statusbits
FCOM	mem	Vergleiche Fließkommazahl aus mem mit st(0), setze entsprechend die Statusbits
FCOM	–	Vergleiche Wert aus st(1) mit st(0), setze entsprechend die Statusbits
FCOMP	st(i)	Vergleiche Wert aus st(i) mit st(0), setze entsprechend die Statusbits, inkrementiere Stapelzeiger
FCOMP	mem	Vergleiche Fließkommazahl aus mem mit st(0), setze entsprechend die Statusbits, inkrementiere Stapelzeiger
FCOMP	–	Vergleiche Wert aus st(1) mit st(0), setze entsprechend die Statusbits, inkrementiere Stapelzeiger
FCOMPP	–	Vergleiche Wert aus st(1) mit st(0), setze entsprechend die Statusbits, inkrementiere Stapelzeiger doppelt
FICOM	mem	Vergleiche ganzzahligen Wert aus mem mit st(0), setze entsprechend die Statusbits
FICOMP	mem	Vergleiche ganzzahligen Wert aus mem mit st(0), setze entsprechend die Statusbits, inkrementiere Stapelzeiger
FTST	–	Vergleiche Wert in st(0) mit 0, setze entsprechend die Statusbits
FXAM	–	Untersuche Wert in st(0), setze entsprechend die Statusbits

Tabelle 4. Alle Assemblerbefehle des Arithmetikprozessors 8087 mit ihrer exakten Bedeutung

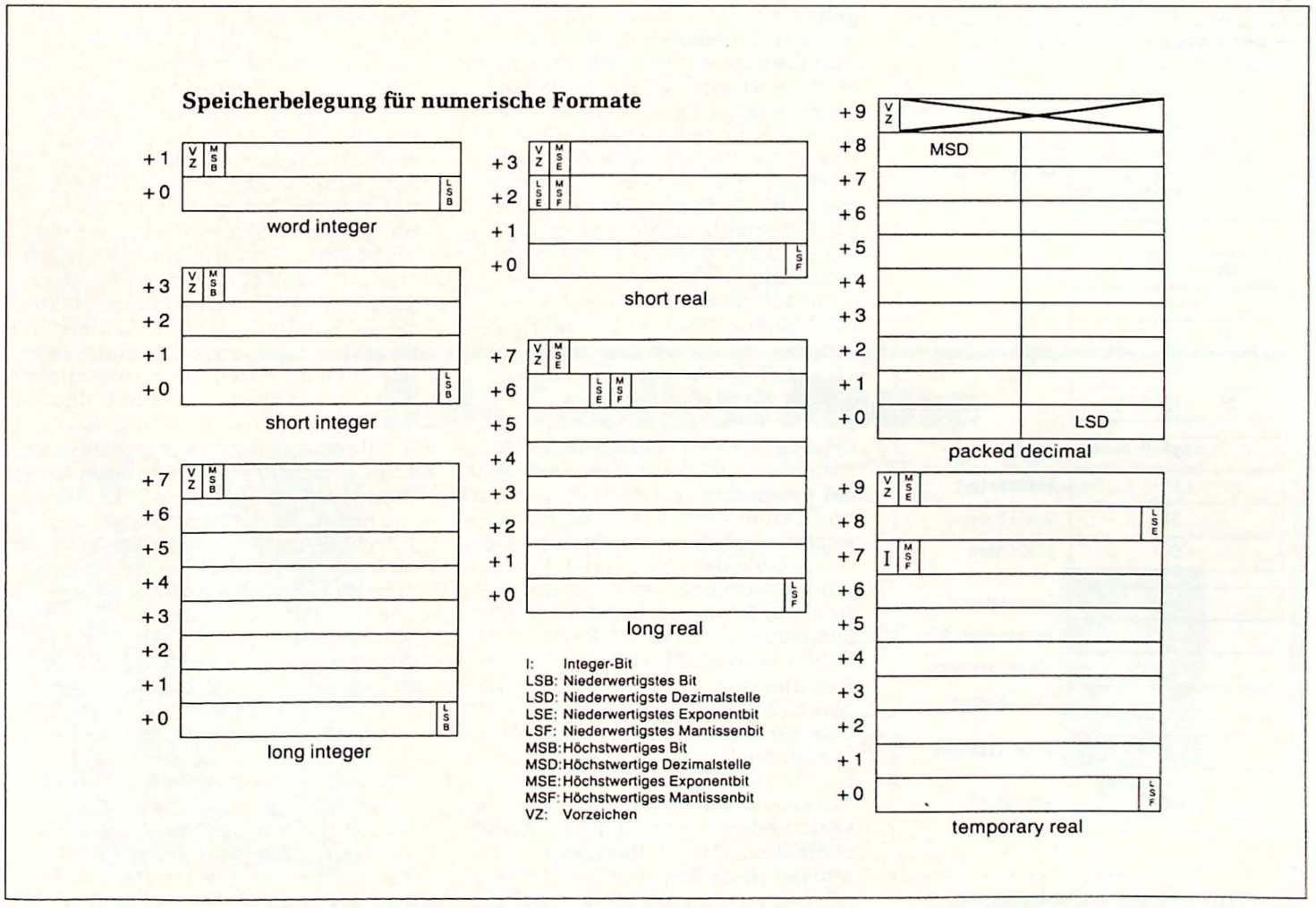


Bild 2. Auf diese Weise werden numerische Werte in verschiedenen Formaten im Arbeitsspeicher des PC abgelegt

Der Registersatz des 8086/80286-Prozessors

Jeder Mikroprozessor besitzt eine Reihe von Registern, in denen der Programmierer Daten oder Zwischenergebnisse ablegt. Bei einem Prozessor der Famile 80x86 gibt es neben den allgemeinen Registern zudem noch Segmentregister, die zur Bildung von vollständigen Adressen benötigt werden.

Ein »Flipflop« ist eine elektronische Schaltung, die eine von zwei verschiedenen Spannungen halten kann – entweder eine niedere, die der Prozessor als »null« oder »unwahr« interpretiert, oder eine hohe, die er als »eins« oder »wahr« annimmt. Ein Register ist ein Satz von 8 oder 16 dieser Flipflops.

Der Assemblerprogrammierer kann einen Prozessor als einen Satz von Registern ansehen, deren Inhalte sich mit dem Befehlssatz auf ganz unterschiedliche Weise modifizieren läßt. Die Register sind für den Programmierer also sehr bedeutende Komponenten des Mikroprozessors, denn in ihnen übergibt er zum Beispiel die Operanden für arithmetische Operationen, oder er speichert temporäre Ergebnisse. Die Register sind sozusagen die Variablen der Assemblersprache.

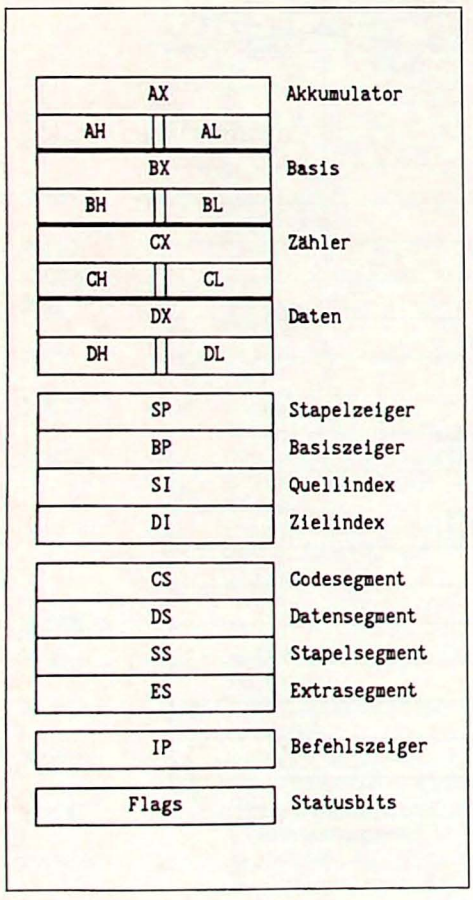


Bild 1. Die Register des 8086/80286-Prozessors

Zwar dienen die Register nur als temporäre Speicherplätze, deren Inhalt zur dauerhaften Speicherung in eine beliebige Speicherzelle kopiert wird, doch dafür erfolgt der Zugriff auf die Register mit einer geradezu atemberaubenden Geschwindigkeit.

Die folgenden Informationen treffen sowohl für den 8086/8088, aber auch für die Prozessoren 80286 und 80386 zu; auf entsprechende Unterschiede wird bei Bedarf hingewiesen.

Die insgesamt 14 Register des 8086 lassen sich ganz grob in allgemeine Register und in Register für besondere Aufgaben unterteilen (Bild 1). Eine weitergehende Unterteilung kennt Datenregister, Zeiger- und Indexregister, Segment- und Kontrollregister.

Als allgemeine oder Datenregister gelten AX, BX, CX und DX, die jeweils 16 Bit breit sind. Dabei läßt sich die obere Hälfte (das höherwertige Byte) und die untere Hälfte (das niederwertige Byte) auch einzeln ansprechen. So teilt sich das AX-Register in die 8-Bit-Register AL (niederwertiges Byte) und AH (höherwertiges Byte). Es liegt an Ihnen, wie Sie die Datenregister verwenden, entweder als 8-Bit- oder als 16-Bit-Register beziehungsweise erst auf die eine, dann auf die andere Weise.

Die Vorteile dieser Registeraufspaltung werden Sie erkennen, wenn Sie an ASCII-Zeichen oder an Zahlen kleiner als 256 denken.

Wie der Name schon andeutet, sind die allgemeinen Register nicht für bestimmte Funktionen reserviert, und der Programmierer darf sie nach Lust und Laune verwenden. Allerdings setzt die Arbeitsweise einiger Assemblerbefehle der völlig freien Verwendung Grenzen. So erwartet zum Beispiel der Loop-Befehl für Schleifen, daß namentlich das CX-Register als Zähler verwendet wird.

Aus diesem Grund und im Interesse übersichtlicher Programme ist es üblich, für bestimmte Anwendungen bestimmte Register zu benutzen. Dabei dient das AX-Register meist dem Austausch beliebiger Daten zwischen verschiedenen Registern und Speicherzellen, das BX-Register als Basisregister (Base Register) bei der Adressierung von Speicherzellen, das CX-Register – wie schon gesagt – als Zähler für Schleifen (Counter Register) und das DX-Register (Data Register) für allgemeine Daten. Das DX-Register hat nur einen speziellen Verwendungszweck – es enthält die Nummer des Ein-/Ausgabekanals bei entsprechenden Port-Instruktionen. Das AX-Register wird auch häufig als Akkumulator bezeichnet. Bei älteren Prozessoren war dieses Register das einzige, in dem arithmetische Operationen durchgeführt wurden und in dem arithmetische Resultate über mehrere Befehle »akkumuliert« werden konnten.

Neben den allgemeinen Registern gibt es auch einige Register, die für bestimmte Aufgaben reserviert sind.

Der 8086 kennt Daten-, Zeiger-/Index-, Segment- und Kontrollregister

Die Gruppe der Zeiger- und Indexregister besteht aus vier 16 Bit breiten Registern: dem Basiszeiger BP (Base Pointer), dem Stapelzeiger SP (Stack Pointer), dem Quellindexregister SI (Source Index) und dem Zielindexregister DI (Destination Index).

Für Operationen mit Zeichenketten oder Datenblöcken im Speicher des PC, also zum Beispiel zum Suchen oder Kopieren bestimmter Zeichenfolgen, dienen die beiden Indexregister DI und SI. Zwar lassen sie sich auch als allgemeine Register einsetzen – für die Zeichenkettenoperationen sind die Indexregister jedoch obligatorisch. Der Basiszeiger dient meist dazu, um auf Elemente des Stapels zuzugreifen oder den Stapelzeiger SP zwischenzuspeichern. Letzterer enthält die Adresse des zuletzt auf dem Stapel gespeicherten Datenwortes und wird durch jedes zusätzliche Element um zwei verkleinert, denn der Stapel wächst in absteigender Richtung.

Zum Verständnis der vier Segmentregister CS (Code Segment), DS (Data Segment), ES (Extra Segment) und SS (Stack Segment) ist es notwendig, sich mit der Art der Speicheradressierung beim 8086 vertraut zu machen. Damit hat es folgende Bewandtnis: Um einen 1 MByte großen Adreßraum anzusprechen, sind 20 Adreßleitungen notwendig – die für die Adressierung der Speicherzellen zuständigen Register sind aber nur 16 Bit breit. Daher wird eine 20-Bit-Adresse in »Segment« und »Offset« aufgeteilt. Der Offset stellt dabei die untere Hälfte der Adresse dar und entstammt dem jeweiligen 16-Bit-Register. Dazu wird der Inhalt eines der vier Segmentregister als Segment addiert, nachdem es mit 16 multipli-

ziert worden ist. Ein Beispiel verdeutlicht dies in hexadezimaler Schreibweise: Der Offset der Adresse sei 1234, und im Segmentregister soll 5678 stehen. Dann addiert sich eine 20-Bit-Speicheradresse gemäß

01234 + 56780 = 567B4

Welches der vier Segmentregister bei der Berechnung einer 20-Bit-Adresse Verwendung findet, ist von den verwendeten Assemblerbefehlen abhängig. So wird die vollständige Adresse des Stapels immer aus dem SP-Register (Offset) und dem SS-Register (Segment) gebildet. Andere Register lassen sich zusammen mit einem beliebigen Segmentregister verwenden. Dazu dient ein sogenanntes »Segment Override Prefix«. Dies ist ein 1-Byte-Assemblerbefehl, der das für die nächste Operation verwendete Segmentregister vorschreibt.
Die Kontrollregister des 8086/8088 enthalten den IP (»Instruction Pointer«) und das Statusregister PSW (»Processor Status Word«). Der Befehlszeiger IP enthält die Adresse des nächsten auszuführenden Befehls, der immer im Zusammenhang mit dem Segmentregister CS gebraucht wird. Das Register IP kann nicht als allgemein verwendbares Register benutzt und daher auch nicht mit einem Mov-Befehl verändert werden. Sobald ein Befehlsbyte aus dem Speicher gelesen wurde, erhöht sich der Inhalt des IP-Registers um eins. Bei Sprungbefehlen oder Unterprogrammaufrufen ändert sich auch entsprechend das IP-Register.
Das Statusregister (oder Flagregister) ist eine Sammlung von einzelnen Statusindikatoren – einem Armaturenbrett eines Autos nicht unähnlich, in dem die einzelnen Warnlämpchen

Das Statusregister des 8086

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	SF	ZF		AF		PF		CF

Das Statusregister des 80286

15	14	13 12	11	10	9	8	7	6	5	4	3	2	1	0
	NT	I/O PL	OF	DF	IF	TF	SF	ZF		AF		PF		CF

Legende:

Abkürzung	englische Bezeichnung	deutsche Bezeichnung
CF	Carry Flag	Übertragskennzeichen
PF	Parity Flag	Paritätskennzeichen
AF	AL-Carry-Flag	Hilfsübertragskennzeichen für AL
ZF	Zero Flag	Nullkennzeichen
SF	Sign Flag	Vorzeichenbit
TF	Trap Flag	Einzelschrittkennzeichen
IF	Interrupt-enable Flag	Unterbrechungsfreigabekennzeichen
DF	Direction Flag	Richtungskennzeichen
OF	overflow flag	Überlaufkennzeichen
Nur 80286/80386 im Protected Mode:		
I/O PL	I/O Privilege Level	Maximale Zugriffsrechte für E/A-Operationen
NT	Nested Task	Kennzeichen für verschachtelte Tasks

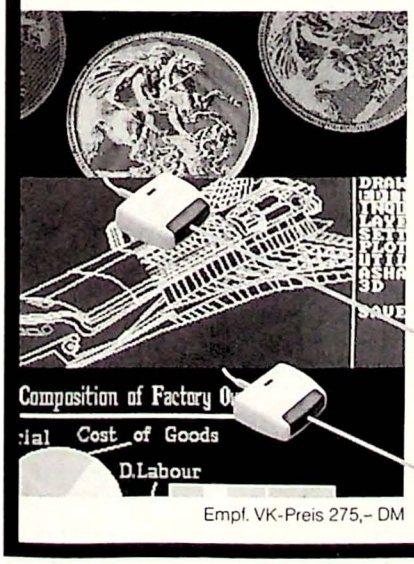
Bild 2. Das Statusregister des 8086/80286-Prozessors

»an« oder »aus« geschaltet sind. Bild 2 zeigt seinen Aufbau. Dabei dienen sechs Bit dazu, den Zustand der letzten arithmetischen Operation anzuzeigen.

Die übrigen Bits des Statusregisters können vom Programmierer durch entsprechende Assembleranweisungen gezielt gesetzt oder gelöscht werden. Über sie läßt sich etwa die Behandlung von Interrupts unterbinden, ein spezieller Einzelschrittmodus zum Testen von Programmen einschalten oder die Richtung bei Zeichenkettenoperationen festlegen. Beim 8086/8088 sind die Bits 15 bis 12 ohne Bedeutung, während sie beim AT zusätzliche Kennzeichen für den multitaskingfähigen »Protected Mode« enthalten.

(Martin Hepp/al)

BMC Cordless Mouse



Empf. VK-Preis 275,– DM

Für PCs (AT/XT und Compatible sowie vergleichbare Leistungsklassen

- Infrarot-Signalübertragung.
- Volle Kompatibilität zur »Microsoft«-Mouse und »Mouse System PC Mouse« wählbar über mitgeliefertes Treiberprogramm.
- Anwendbar unter allen Programmen, die eine Mouse unterstützen.
- Geeignet für Tabellenkalkulationen, Textverarbeitung, Desktop Publishing, CAD-CAM, usw.
- Software incl. Mousetreiber, Testprogramm und Programm zur Erstellung eigener mousegesteuerter Menüs für Standardpakete, die keine Mouse unterstützen.
- Präzise Bewegungen des Cursors auf dem Bildschirm durch eine spezielle 'Kriechstaste'.
- Antistatische silikonüberzogene Stahlkugel.
- Niedrige Reibung aufgrund einer Teflonbeschichtung.

Technische Daten:

8 Bit, 12 MHz, CMOS CPU
Auflösung 200 dpi, im Normalzustand 10-800 dpi möglich, wenn seitliche Taste betätigt wird.
Reichweite 1,5 m, Arbeitswinkel 90°
Geschwindigkeitsübertr.: 600 mm/sec.
Bedienung über drei programmierbare Tasten
Alle Tasten mit fühl- und hörbarem Druckpunkt
Standard RS 232 Schnittstellen

Lieferung ausnahmslos an Großabnehmer und Fachhandel

BMC
INTELLIGENT TECHNOLOGY

Mühlenweg 2
4724 Wadersloh 3 - Diestedde
Tel. (02520) 581
Telex 89562, Telefax 583

Die Speicheradressierung des 8086/8088

Für die Programmierung in Maschinensprache ist die Art und Weise, wie der Prozessor den Arbeitsspeicher adressiert, von grundlegender Bedeutung, denn der Programmierer ist hier in der Regel für den Zugriff auf den Arbeitsspeicher selbst verantwortlich. Es ist daher wichtig zu wissen, nach welchem Prinzip die Prozessoren der 80x86-Familie den Speicher adressieren und welche Bedeutung dies für den Aufbau eines Assemblerprogramms hat.

Die Intel-Prozessoren 8086 und 8088 verfügen über 20 Adreßleitungen, die über den Adreßbus mit dem Arbeitsspeicher verbunden sind. Mit 20 Adreßleitungen lassen sich insgesamt 2^{20} gleich 1048576 Byte (1 MByte) adressieren. Allerdings verfügt der 8086/8088 intern nur über Register mit einer »Breite« von 16 Bit. Würde zur Adreßberechnung lediglich ein Register zugrunde gelegt, ließe sich nur ein Arbeitsspeicher von 2^{16} gleich 65536 Byte adressieren. Um dennoch die gesamte Kapazität nutzen zu können, haben sich die Entwickler bei Intel etwas Besonderes einfallen lassen: Jede Adresse wird stets aus zwei Komponenten zusammengesetzt; einem Segmentanteil und einem sogenannten Offset, beide 16 Bit breit. Der Segmentanteil befindet sich in einem der vier Segmentregister CS, DS, ES oder SS, während es sich bei dem Offset beispielsweise um eine 16-Bit-Konstante oder um den Inhalt eines Registers (zum Beispiel des BX-Registers) handeln kann.

Immer, wenn eine Speicheradresse zu bilden ist, multipliziert der Prozessor zunächst den Segmentanteil mit 16. Dies ergibt ein Ergebnis mit 20 binären Stellen. Die Multiplikation einer Dualzahl mit zwei entspricht bekanntlich dem Verschieben der gesamten Zahl um eine Stelle nach links. Die neu entstehende Ziffer am rechten Ende wird mit einer Null aufgefüllt. Eine Multiplikation mit 16 kommt daher einem viermaligen Verschieben nach links gleich. Als nächstes wird der 16-Bit-Offset zu dem Ergebnis der Multiplikation addiert, woraus die endgültige 20-Bit-Adresse resultiert. Dieser Wert, der auch als physikalische Adresse bezeichnet wird, kennzeichnet nun genau eine Speicherzelle des Arbeitsspeichers. Das Bild veranschaulicht dieses Prinzip an einem Beispiel. Obwohl die physikalische 20-Bit-Adresse innerhalb des Prozessors, genauer gesagt in der Bus-schnittstelleneinheit, gebildet wird, ist sie für den Programmierer nicht zugänglich.

Diese spezielle Art der Adreßberechnung hat zahlreiche direkte und indirekte Konsequenzen. Die offensichtlichste Konsequenz ist, daß sich durch diese Adressierung der gesamte Arbeitsspeicherbereich zwangsläufig in Untereinheiten, die Segmente genannt werden, aufteilt. Physikalisch gesehen handelt es sich bei dem Arbeitsspeicher eines PC zwar um einen durchgehenden Speicherbereich, für den Prozessor stellt er sich jedoch in einzelne Segmente aufgeteilt dar. Ein Segment ist daher nichts anderes als ein bestimmter Bereich des Arbeitsspeichers.

Die Startadresse eines Segments wird stets durch den Inhalt eines der vier Segmentregister festgelegt. Jedes Byte innerhalb dieses Segments wird über den Offset adressiert. Da es sich bei dem Offset immer um einen 16-Bit-Wert handelt, ist die Größe eines Segments auf 65536 Byte begrenzt. Eine weitere Konsequenz ist, daß ein Segment nur bei einer Adresse beginnen darf, die ohne Rest durch 16 teilbar ist (Paragraphenadresse).

Die Erklärung hierfür ist einfach: Stellen Sie sich in einem Segmentregister eine beliebige 16-Bit-Zahl vor. Multiplizieren Sie diese Zahl mit 16, dann ergibt sich ein 20-Bit-Wert mit vier Nullen am rechten Ende (den niederwertigsten Stellen). Das heißt, daß die Startadresse eines Segments in jedem Fall glatt durch 16 teilbar ist.

Schließlich beeinflußt auch die Aufteilung in Segment- und Offsetanteil die Darstellungsweise einer Adresse. Adressen werden allgemein im Format »Segment:Offset« angegeben. So bedeutet B000:0012hex, daß die Adresse aus dem Segmentanteil B000hex und dem Offsetanteil 0012hex besteht (»hex« deutet an, daß es sich um Hexadezimalzahlen handelt, wobei eine Stelle dieses Zahlensystems vier Stellen des binären Systems entspricht).

Für die Adresse B000:0012hex hätten Sie genausogut auch B0012hex schreiben können (B000hex mal 16 plus 0012hex), allerdings ist diese Darstellungsweise unüblich.

Wie viele Segmente lassen sich nun beim 8086/8088 maximal definieren? Da ein Segment an jeder 16ten Adresse beginnen darf, kann es maximal 1048576/16 = 65536 verschiedene Segmente geben (die Größe eines Segments spielt dabei keine Rolle, denn es können sich durchaus mehrere Segmente überlappen). Gleichzeitig lassen sich aber nur maximal vier Segmente addressieren, da es nur vier verschiedene Segmentregister gibt.

Die Adresse einer Speicherzelle läßt sich vielfältig darstellen

Auch wenn der Sachverhalt für einen Einsteiger dadurch eher komplizierter wird, sei dennoch angemerkt, daß der Assembler die Segmentverteilung großzügiger handhabt. Innerhalb eines Assemblerprogramms lassen Sie über den sogenannten Ausrichtungstyp (»align type«) Segmente im Prinzip an jeder beliebigen Adresse beginnen. Falls es sich bei der Startadresse eines Segments nicht um eine Paragraphenadresse handelt, müssen Sie den Offset innerhalb des betreffenden Segments entsprechend anpassen. Bei Verwendung der vereinfachten Segmentanweisungen (beispielsweise »code« oder »data«) ist der Ausrichtungstyp standardmäßig festgelegt, bei Verwendung der Anweisung »segment« zur Definition eines Segments läßt er sich frei wählen.

Nach soviel Theorie erfahren Sie nun anhand einiger Beispiele, wie die Bildung der physikalischen Adresse in der Praxis erfolgt.

Speicherzugriff	Standardsegment	Alternative	Offset
Befehl lesen	CS	keine	IP
Datenoperation	DS	CS, ES, SS	EA
Stapeloperation	SS	keine	SP
BP als Basis	SS	CS, DS, ES	EA
BX als Basis	DS	CS, ES, SS	EA

EA = effektive Adresse

Tabelle. Die standardmäßige Segmentzuordnung

Betrachten Sie zunächst einmal folgenden Befehl

```
mov ax,dx
```

der den Inhalt des DX-Registers in das AX-Register kopiert. Bei der Ausführung dieses Befehls wird nicht auf den Arbeitsspeicher zugegriffen, also spielt die Adreßberechnung keine Rolle. Ganz anders sieht es bei dem nächsten Befehl aus, der auf einen im Arbeitsspeicher gespeicherten Wert zugreift. Zunächst soll dazu im Datensegment eine Variable mit dem Namen »zahl« definiert werden:

```
.data
zahl dw 1000
```

Der symbolische Name »zahl« steht für die Offsetadresse zweier Speicherzellen innerhalb des Datensegments (welches Sie durch die Anweisung ».data« definieren), die mit dem Wert 1000 initialisiert werden. Aus diesem Grund darf man »zahl« als Variable auffassen, da der Inhalt der Speicherzellen veränderbar ist. Mit dem nächsten Befehl laden Sie den Inhalt der Variablen »zahl« in das AX-Register:

```
mov ax,zahl
```

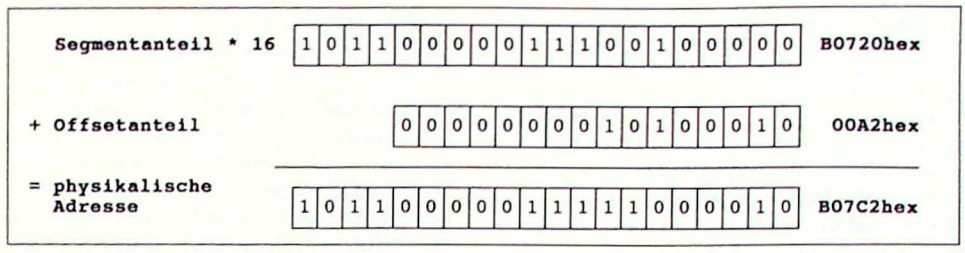
Wie Sie inzwischen erfahren haben, steht »zahl« nur für den Offset einer Speicherzelle. Wie sieht es aber mit dem Segmentanteil aus? Dazu muß man wissen, daß bei dieser Art des Speicherzugriffs der Prozessor davon ausgeht, daß sich der Segmentanteil im DS-Register befindet. Genaugenommen wird also der Inhalt der Speicherzelle »DS:zahl« in das AX-Register geladen.

Segmentregister gewährleisten, daß Datenzugriffe korrekt erfolgen

Damit diese Art der Adreßberechnung funktioniert, muß aber sichergestellt sein, daß im DS-Register der richtige Wert enthalten ist. Dies läßt sich beispielsweise mit der Befehlssequenz

```
mov dx,@data
mov ds,dx
```

erreichen, die das DS-Register mit der Adresse des Datensegments lädt. Das DS-Register zeigt nun auf den Beginn des Datensegments, während über den Offset (beispielsweise »zahl«) jedes Byte innerhalb des Datensegments ansprechbar ist. »@data« ist eine Konstante, die der Assembler (Microsoft Makroassembler ab Version 5.0) zur Verfügung stellt, um die Adresse des Datensegments innerhalb eines Programms verwenden zu können. Der Umweg über das DX-Register ist nur deswegen erforderlich,



So setzt sich eine physikalische Adresse zusammen

weil Segmentregister keine direkten Zuweisungen zulassen. Übrigens ist bei Com-Dateien eine Initialisierung des DS-Registers nicht notwendig, da beim Laden einer solchen Datei die Adresse des (einzigen) Segments automatisch in das DS-Register eingetragen wird.

Denken Sie auch daran, daß Segmentadressen innerhalb eines Assemblerprogramms nur symbolische Größen darstellen. Absolute Adressen vergibt weder der Assembler noch der Linker – sie werden erst beim Laden des Programms in den Arbeitsspeicher festgelegt.

Natürlich lassen sich über einen Befehl wie »mov ax,zahl« auch Daten ansprechen, die über das ES-Register adressiert sind. In diesem Fall muß aber der Quelladresse der Segment-Override-Operator »:« vorangehen:

```
mov ax,es:zahl
```

Dieser Operator sorgt zusammen mit der Angabe des Segmentregisters dafür, daß vor den Opcode des Mov-Befehls das passende Präfix (in diesem Fall »es:«) assembliert wird. Trifft der Prozessor bei der Programmausführung auf dieses Präfix (es handelt sich um ein einzelnes Byte), dann »weiß« er, daß sich der Segmentanteil der Speicheradresse nicht im DS-Register, sondern diesmal im ES-Register befindet. Der Segment-Override-Operator dient allgemein dazu, Standardsegmentzuordnungen bei Bedarf aufzuheben. Allerdings gibt es bei der Auswahl der Segmentregister bestimmte Einschränkungen (siehe Tabelle).

Nicht nur bei den Mov-Befehlen spielt die implizite Registerzuordnung eine Rolle. Generell gibt es für jeden auf den Speicher zugreifenden Befehl ein Segmentregister, welches im Standardfall – das heißt, wenn nichts anderes vereinbart ist – den Segmentanteil zur Verfügung stellt. Dazu gehören auch die Befehle »push« und »pop«. Bekanntlich transportiert ein Befehl wie

```
push ax
```

den Inhalt des AX-Registers auf den Stapel. Da es sich bei diesem aber um einen Teil des Arbeitsspeichers handelt, wird auch hier eine Speicheradressierung benötigt. Anders als bei den Mov-Befehlen ist der Offset für die Zieladresse im SP-Register enthalten, während sich der Segmentanteil im SS-Register befindet. Da sich Befehle wie »push« oder »pop« ausnahmslos auf den Stapelbereich beziehen, dürfen Sie hier keinen Segment-Override-Operator verwenden.

Nicht nur Daten, sondern auch Befehle holt sich der Prozessor aus dem Speicher

Bislang wurde lediglich der Zugriff auf Daten im Arbeitsspeicher betrachtet. Aber auch bei der Programmausführung muß der Prozessor auf Daten zugreifen, in diesem Fall auf die Opcodes der Maschinenbefehle. Der als nächstes aus dem Speicher zu lesende Befehl wird stets durch das Registerpaar CS:IP adressiert, wobei das CS-Register den Segment- und das IP-Register den Offsetanteil zur Verfügung stellt.

Welche Rolle spielt die Speicheradressierung für ein Assemblerprogramm? Wie Sie wahrscheinlich wissen, stellt der Assembler Anweisungen zur Verfügung, mit denen sich der Beginn und das Ende eines Segments festlegen lassen. So leitet beispielsweise ».code« den Beginn des Programmsegments oder ».data« den Beginn des Datensegments ein, während ein eventuell vorangehendes Segment gleichzeitig beendet wird. Diese Anweisungen wären nicht notwendig, wenn das Programm nur aus einem einzigen Segment bestehen würde (beim Turbo Assembler entspräche dies dem Speichermodell »tiny«). In diesem Fall könnten Sie sich sämtliche Segmentdefinitionen sparen, da alle Befehle beziehungsweise Daten in ein und dasselbe Segment eingetragen würden. In der Praxis sieht es aber anders aus.

So verwenden die meisten Programme mindestens drei Segmente: ein Stacksegment, ein Datensegment und ein Programmsegment. Das heißt nun nicht etwa, daß jedes Segment den vollen Bereich von 64 KByte belegen muß, vielmehr belegt ein Segment nur soviel Speicherplatz, wie ihm innerhalb des Assemblerprogramms zugewiesen wird. Betrachten Sie dazu einmal das folgende Programm-

gerüst:

```
.model small
.stack 256
.data
feld dw 10,20,30
.code
mov dx,@data
mov ds,dx
end
```

Zunächst wird durch die Anweisung ».model« das Speichermodell ausgewählt. Dieses legt fest, ob der Datenbeziehungsweise Codebereich aus einem oder mehreren Segmenten bestehen darf. Im Beispiel wird das Speichermodell »small« gewählt, in dem sowohl der Daten- als auch der Codebereich jeweils nicht größer als 64 KByte werden können. Nachdem Sie diesen notwendigen Formalismus erledigt haben, definieren Sie die einzelnen Segmente. Üblicherweise beginnt man dabei mit dem Stapelsegment. Die Größe des Stacksegments ist leicht ersichtlich – es werden genau 256 Byte reserviert. Die Größe des folgenden Datensegments hängt von der Art und Anzahl der darin befindlichen Datenanweisungen ab. In unserem Fall werden drei Wörter reserviert, die insgesamt sechs Byte belegen. Die Größe des Codesegments beträgt im obigen Beispiel genau fünf Byte, da die Opcodes der beiden Mov-Befehle eben fünf Byte umfassen. Über die Größe der in einem Programm verwendeten Segmente gibt Ihnen auch die Map-Datei Aufschluß, die Ihnen der Linker (auf Wunsch) erzeugt.

Die tatsächliche Größe des Programms im Arbeitsspeicher beziehungsweise der Programmdatei läßt sich aber aus der Größe der einzelnen Segmente nicht direkt berechnen. Zum einen können zwischen zwei benachbarten Segmenten im Arbeitsspeicher durchaus Lücken existieren, die damit zusammenhängen, daß manche Segmente nur an Wort- oder Paragraphenadressen beginnen (Stichwort: Ausrichtungstyp). Dadurch belegt ein Programm mehr Byte im Arbeitsspeicher, als es eigentlich der Summe der einzelnen Segmentgrößen entsprechen würde. Zum anderen ist bei Programmdateien der Kopf einer Exe-Datei zu berücksichtigen, der innerhalb der Programmdatei ebenfalls einige Byte belegt.

Nachdem Sie das Prinzip der Adressierung kennengelernt haben, nun noch zu den einzelnen Adressierungsarten. Den Prozessoren 8086 und 8088 stehen folgende Adressierungsarten zur Verfügung:

- Registeradressierung
- unmittelbare Adressierung
- direkte Adressierung
- indirekte Adressierung
- Basisadressierung
- indizierte Adressierung
- basisindizierte Adressierung

Die Adressierungsarten lassen sich grob in zwei Gruppen einteilen – Register- und Speicheradressierung. Die Registeradressierung haben Sie bereits kennengelernt. Hier werden lediglich Daten innerhalb des Prozessors transportiert; der Arbeitsspeicher muß nicht angesprochen werden. Dies gilt auch für die unmittelbare Adressierung, falls ein Register mit einem Wert zu laden ist. Der Befehl »mov ax,100«, der das AX-Register mit dem Wert 100 lädt, fällt in diese Kategorie.

Unterschiedliche Adressierungsarten erlauben eine flexible Programmierung

Die zweite Gruppe der Adressierungsarten ist die Speicheradressierung. Hier spielt die effektive Adresse eine wichtige Rolle. Sie entspricht dem Offsetanteil der physikalischen Adresse, es handelt sich also um eine vorzeichenlose 16-Bit-Zahl. Die einzelnen Speicheradressierungsarten unterscheiden sich einzig und allein dadurch, auf welche Weise die effektive Adresse gebildet wird. Der einfachste Fall liegt vor, wenn ein Register mit dem Inhalt einer Speicherzelle geladen werden soll:

```
mov ax,summe
```

Die effektive Adresse wird hierbei durch den Offset der Speicherstelle mit dem symbolischen Namen »summe« gebildet. Da die Adresse bereits in dem Befehl enthalten ist, bezeichnet man sie als direkte Adressierung. Bei der indirekten Adressierung befindet sich die effektive Adresse dagegen in einem der Register BP, BX, SI oder DI. Hier ein kleines Beispiel:

```
mov ax,[bx]
```

Dieser Befehl lädt den Inhalt einer Speicherzelle, die durch den Inhalt des BX-Registers adressiert ist, in das AX-Register. Beachten Sie, daß die effektive Adresse bei der Assemblierung des Programms noch gar nicht bekannt ist. Diese Art der Adressierung findet auch bei den indirekten Sprungbefehlen oder Unterprogrammaufrufen statt. Allerdings dürfen Sie hier alle allgemeinen Register verwenden. So verzweigt der Befehl

```
jmp ax
```

zu der Routine, deren Offsetadresse im AX-Register enthalten ist, während der Befehl

```
jmp [bx]
```

zu der Routine springt, deren Adresse in der Speicherzelle enthalten ist, die durch das BX-Register adressiert wird.

Ein wenig komfortabler ist die Basisadressierung, da hier zusätzlich zum Inhalt eines Registers noch eine Konstante angegeben werden darf. Auf diese Weise lassen sich Felder oder Tabellen innerhalb einer Schleife mit einem einzigen Mov-Befehl adressieren. Zuerst wird ein kleines Feld mit zehn Elementen angelegt:

```
feld db 10 dup (?)
```

Anschließend erhält jedes Element einen Wert zugewiesen:

```
mov al,32
mov bx,9
Schleife:
mov feld[bx],al
dec bx
jnz Schleife
```

Bei jedem Durchlauf der Schleife wird die effektive Adresse aus der Summe des Offsets von »feld« und dem Inhalt des BX-Registers gebildet, welches wiederum bei jedem Durchlauf um eins erniedrigt wird. Zur Basisadressierung läßt sich entweder das BX- oder das BP-Register verwenden. Beachten Sie, daß sich der Segmentanteil bei Verwendung des BX-Registers standardmäßig im DS-Register, bei Verwendung des BP-Registers dagegen im SS-Register befindet.

Die indizierte Adressierung entspricht im Prinzip der Basisadressierung; der einzige Unterschied besteht in der Verwendung der Register DI und SI anstelle von BX und BP.

Die komfortabelste Adressierungsart stellt die basisindizierte Adressierung dar. Sie läßt sich beispielsweise zur Adressierung zweidimensionaler Felder einsetzen. Hier setzt sich die effektive Adresse aus drei Komponenten zusammen: einem Basisregister (BX oder BP), einem Indexregister (DI oder SI) und schließlich einem konstanten Zahlenwert (Displacement), wobei letzterer auch entfallen darf. Auch dazu ein Beispiel:

```
mov ax,[bx][di]100
```

Der Offset der angesprochenen Speicherzelle, das heißt die effektive Adresse, berechnet sich dabei folgendermaßen:

Inhalt des BX-Registers
+ Inhalt des DI-Registers
+ 100

Das Schema der Adreßbildung bei den Prozessoren der 80x86-Familie ist zugegebenermaßen schon recht verwirrend. Als Einsteiger hat man seine liebe Müh' und Not, die einzelnen Adressierungsarten und vor allem auch die verschiedenen Begriffe wie etwa effektive Adresse und physikalische Adresse auseinanderzuhalten. Es ist wahrscheinlich auch kein Trost für Sie, daß das Adressierungs-

schema bei den Prozessoren 80286, 80386 und 80486 im sogenannten Protected-Modus noch ein wenig komplizierter ist.

Der Vorteil der Segmentierung: Programme sind im Speicher frei zu verschieben

Dennoch dürfen Sie nicht vergessen, daß die Segmentierung auch Vorteile mit sich bringt. So werden für Sprungbefehle innerhalb des Segments nur zwei Byte für die Sprungadresse benötigt, da in diesem Fall nur der Offset angegeben werden muß. Des weiteren – und hier liegt der entscheidende Vorteil – sind Programme, die lediglich Offsets verwenden, vollkommen positionsunabhängig. Sie können ein Programm also beliebig im Speicher verschieben, ohne irgendwelche Adressen anpassen zu müssen.

Zur Rekapitulation, zwei Dinge sind für das Verständnis der Speicheradressierung bei den 80x86-Prozessoren wichtig:

– Jede Adresse besteht aus einem Segment- und einem Offsetanteil. Der Segmentanteil befindet sich immer in einem der vier Segmentregister. Der Offsetanteil, das heißt die effektive Adresse, läßt sich auf verschiedene Weise bilden. Er kann in einem Basis- oder Indexregister enthalten sein, er kann aber auch direkt als Konstante aufgeführt werden. Die physikalische 20-Bit-Adresse bildet der Prozessor, indem er den Segmentanteil mit 16 multipliziert und den Offsetanteil dazuaddiert.

– Absolute Adressen kommen innerhalb eines Assemblerprogramms so gut wie nie vor. In den meisten Fällen wird der Segmentanteil nur symbolisch über vordefinierte Textkonstanten wie zum Beispiel »@code» oder »@data« verwaltet. Eine konkrete Adresse steht erst beim Laden des Programms fest. Lediglich um den Offsetanteil müssen Sie sich in manchen Fällen selbst kümmern. Doch in der Regel wird auch dieser symbolisch angesprochen, beispielsweise über einen Variablennamen.

(Peter Monadjemi/zi)

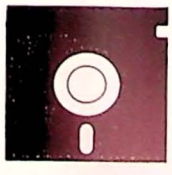
DOS INTERNATIONAL

Zur Ergänzung unseres jungen, engagierten Redaktionsteams suchen wir eine(n) versierte(n)

Hardware-Fachredakteur/in

Sie haben bereits grundlegende Erfahrungen im journalistischen Bereich und fundierte Hardware-, aber auch Softwarekenntnisse. Das Schreiben macht Ihnen Freude, und Sie sind in der Lage, komplizierte Zusammenhänge verständlich darzustellen. Bringen Sie dann noch die Bereitschaft mit, selbständig zu arbeiten, sollten wir uns kennenlernen.

Sind Sie interessiert? Dann senden Sie uns Ihre Bewerbungsunterlagen. Wir freuen uns auf ein persönliches Gespräch.



Redaktionsbüro Everts & Hagedorn
Redaktion DOS International
Hans-Stießberger-Str. 3
8013 Haar/München

Assemblerreferenz: Programmablaufsteuerung

In dieser Übersicht finden Sie alle Befehle, die den normalen, sequentiellen Programmablauf ändern. Befehle zur Ablaufsteuerung sind zum Beispiel Unterprogrammaufrufe, Verzweigungskommandos, Interrupt-orientierte Befehle und Schleifenkommandos.

CALL

»Call procedure«, starte Unterprogramm

```
Syntax: CALL <Unterprogramm>
```

Mit diesem Befehl starten Sie ein Unterprogramm. Der Prozessor setzt die Programmausführung beim ersten Befehl des Unterprogramms fort. Nach Beendigung desselben wird das Hauptprogramm direkt hinter dem Call-Befehl fortgesetzt.
Der Parameter »Unterprogramm« darf das Label einer Near- oder Far-Prozedur (innerhalb oder außerhalb des aktuellen Segments) sein. Ein Assembler generiert je nach verwendetem Operanden (entweder Label oder Register) unterschiedliche Maschinenbefehle für die einzelnen Call-Befehle.
Je nachdem, ob ein Near- oder ein Far-Call vorliegt, legt der Prozessor vor dem Unterprogrammaufruf entweder nur das IP- oder das IP- und das CS-Register auf den Stapel. Erreicht der Computer das Ende eines solchen Unterprogramms (Ret-Anweisung), so wird der Befehlszeiger wieder vom Stapel geholt und das auf den Call-Befehl folgende Kommando ausgeführt.
Beispiel:

```
CALL PROZEDUR ;starte Unterprogramm
...
PROZEDUR PROC NEAR ;Beginn der Prozedur
...
PROZEDUR ENDP ;Ende der Prozedur
```

CLI

»Clear interrupt-enable-flag«, lösche das Interrupt-Flag, sperre Interrupts

```
Syntax: CLI
```

Dieser Befehl löscht das Interrupt-Kennzeichen. Anhand dieses Flags erkennt der Prozessor, ob maskierbare Unterbrechungen gesperrt oder zugelassen sind. Sie veranlassen also durch das CLI-Kommando, daß alle maskierbaren Unterbrechungen nicht mehr ausgeführt werden. Diese Blockade der Interrupts heben Sie durch den STI-Befehl wieder auf.
Beispiel:

```
CLI ;verhindere Interrupts
```

ESC

»Escape«, Signal

```
Syntax: ESC <Opcode>,<Wert>
```

Der Esc-Befehl signalisiert einem Coprozessor, daß auf dem Datenbus Befehle für ihn bereitliegen. Der Hauptprozessor führt die notwendigen Speicherzugriffe aus und stellt den Opcode (Befehl für den Coprozessor) und Zusatzdaten (»Wert«) auf dem Datenbus zur Verfügung. Wie der angeschlossene Coprozessor den Opcode dann verarbeitet, hängt nur von der internen Realisierung des Coprozessors ab.
Beispiel:

```
ESC 12h,AL ;führe Befehl 12hex des Coprozessors aus und stelle den Inhalt des AL-Registers auf dem Datenbus zur Verfügung
```

HLT

»Halt«, Prozessor-Stop

```
Syntax: HLT
```

Dieser Befehl setzt den Prozessor in den Haltezustand. In diesem Zustand tut der Rechner nichts weiter, als den Zustand der Hardware-Leitungen »Reset« und »NMI« zu überwachen. Sind maskierbare Unterbrechungen nicht gesperrt, dann wird der Haltezustand auch verlassen, wenn am Eingang »INTR« ein Signal anliegt.
Beispiel:

```
HLT ;aktiviere Haltezustand
```

INT

»Interrupt«, löse einen Interrupt aus

```
Syntax: INT <Interrupt-Nummer>
```

Durch dieses Kommando löst der Prozessor eine Software-Unterbrechung der angegebenen »Interrupt-Nummer« aus. Bei PCs, ATs, 386er-PCs und PS/2-Modellen dienen Software-Interrupts dazu, auf Funktionen des Betriebssystems zurückzugreifen. Der aktuelle Befehlszeiger, der durch die Register CS und IP bestimmt ist, und das Flag-Register werden auf den Stack gerettet. Dann berechnet der Prozessor aus der Interrupt-Nummer die Adresse des Interrupt-Vektors in der Interrupt-Tabelle ab Speicheradresse 0000:0000h. Hier werden Segment- und Offset-Adresse ausgelesen, und die Bearbeitung des Programms wird an der entsprechenden Stelle fortgesetzt.
Beispiel:

```
MOV DX,OFFSET MSG ;lade Adresse eines Textes
MOV AH,9h ;lade Unterfunktionsnummer
INT 21h ;löse Interrupt 21hex aus
```

INTO

»Interrupt on overflow«, löse einen Interrupt aus, wenn das Overflow-Flag gesetzt ist

```
Syntax: INTO
```

Dieses Kommando löst einen Interrupt aus, falls das Überlaufstatusbit (»Overflow Flag«) gesetzt ist. Ist es nicht gesetzt, geschieht nichts. Die Zieladresse des Interrupts beginnt ab 0000:0010hex (vierter Interrupt). Das Interrupt-Enable-Flag muß ebenfalls gesetzt sein, das heißt, Interrupts sind zugelassen.
Beispiel:

```
INTO ;löse einen Overflow-Interrupt aus
```

IRET

»Return from interrupt«, Rückkehr aus einem Interrupt

```
Syntax: IRET
```

Der IRET-Befehl dient in Prozeduren zur Verarbeitung von Interrupts dazu, wieder an die Stelle im Programm zurückzukehren, an der der Interrupt ausgelöst wurde. Das IRET-Kommando restauriert das Kennzeichenregister sowie das IP- und CS-Register mit den auf dem Stack gespeicherten Werten.
Beispiel:

```
IRET ;kehre zurück ins Hauptprogramm
```

Jcc

»Jump on condition code«, verzweige (springe) unter einer bestimten Bedingung

```
Syntax: Jcc <Sprungziel>
```

Mit diesem Kommando wird zur Adresse »Sprungziel« verzweigt, falls die mit dem Parameter »cc« formulierte Bedingung erfüllt ist. »cc« kann eine der folgenden Buchstabenkombinationen sein:

B	kleiner als
NAE	nicht größer und nicht gleich
C	Carry-Flag gesetzt
CXZ	CX-Register gleich 0
BE	kleiner als oder gleich
NA	nicht größer

E	gleich
Z	bei Null
L	kleiner als
NGE	nicht größer und nicht gleich
LE	kleiner oder gleich
NG	nicht größer
NB	nicht kleiner
AE	größer oder gleich
NC	Carry-Flag nicht gesetzt
NBE	nicht kleiner und nicht gleich
A	größer
NE	nicht null
NZ	nicht null
NL	nicht kleiner
GE	größer oder gleich
NLE	nicht kleiner und nicht gleich
G	größer
NO	kein Überlauf (gelöschtes OF-Flag)
NP	keine Parität
PO	Parität ungerade
NS	kein Vorzeichen
O	Überlauf (gesetztes OF-Flag)
P	Parität
PE	Parität gerade
S	Vorzeichen

Man benutzt den Jcc-Befehl in der Regel nach einem Vergleich, um, abhängig vom Ergebnis des Vergleichs, zu einem bestimmten Sprungziel zu verzweigen. Da »Sprungziel« im Maschinenbefehl als 8-Bit-Entfernungswert codiert ist, darf das Ziel nur maximal 126 Byte vor oder 127 Byte nach dem Jcc-Befehl liegen.
Beispiel:

```
MOV AX,100 ;lade AX mit 100
CMP AX,SPEICHER ;vergleiche mit »SPEICHER«
JG GROSS ;größer? wenn ja, dann gehe zu »GROSS«
JB KLEIN ;kleiner? ja, gehe zu »KLEIN«
... ;weder noch, also gleich
```

JMP

»Jump unconditionally«, unbedingter (sofortiger) Sprung zu einer anderen Adresse

Syntax: JMP <Sprungziel>

Mit diesem Befehl wird die Programmausführung an der Stelle »Sprungziel« fortgesetzt. Der Befehl ist an keinerlei Bedingungen gebunden. Weil »Sprungziel« im Maschinenbefehl als 8-Bit-Entfernungswert codiert ist, darf das Ziel nur maximal 126 Byte vor oder 127 Byte nach dem JMP-Befehl liegen.
Beispiel:

```
MOV AX,200h ;lade 200hex nach AX
JMP MARKE ;und springe zu »Marke«
```

LOCK

»Lock the bus«, blockiere den Prozessorbus

Syntax: LOCK

»Lock« ist ein sogenanntes »Ein-Byte-Präfix« und wird einem anderen Befehl vorangestellt. Es bewirkt, daß der Prozessor für die Dauer dieses Befehls keine Unterbrechungen, zum Beispiel von anderen Peripheriegeräten (I/O-Einheiten), zuläßt.
Beispiel:

```
LOCK MOV AX,SPEICHER ;Übertragen des Inhaltes einer Speicherstelle ins AX-Register ohne Unterbrechungen
```

LOOP

»Loop if CX not zero«, führe eine Schleife aus, wenn das CX-Register (Schleifenzähler) ungleich 0 ist

Syntax: LOOP <Sprungziel>

Mit dem LOOP-Befehl wird veranlaßt, daß zunächst das CX-Register um eins dekrementiert (heruntergezählt) wird. Falls es dann noch nicht den Wert Null besitzt, wird die Bearbeitung des Programms an der Marke »Sprungziel« fortgesetzt, ansonsten der nächste Befehl hinter dem LOOP-Kommando ausgeführt. Man verwendet das LOOP-Kommando, um auf einfache Art und Weise Schleifen zu programmieren. Sie laden dazu den Maximalwert der Schleife in das CX-Register und führen am Endes des Programmblocks innerhalb der Schleife ein LOOP-Kommando aus. Weil »Sprungziel« im Maschinenbefehl als 8-Bit-Entfernungswert codiert ist, darf es nur maximal 126 Byte vor oder 127 Byte hinter dem LOOP-Befehl liegen.
Beispiel:

```
MOV CX,100 ;übertrage 100 nach CX
XOR AL,AL ;lösche AL
MARKE MOV ES:[CX],AL ;lösche indiziertes Byte
LOOP MARKE ;und bleibe in Schleife
```

LOOPE

»Loop while equal«, führe Schleife aus, solange CX ungleich Null und das Zero-Flag gesetzt ist

Syntax: LOOPE <Sprungziel>

Mit dem LOOPE-Befehl veranlassen Sie den Prozessor, zunächst das CX-Register um eins zu dekrementieren. Falls es dann noch nicht den Wert Null besitzt, wird die Bearbeitung des Programms an der Marke »Sprungziel« fortgesetzt, wenn zusätzlich das Nullkennzeichen (»Zero-Flag«) gesetzt ist. Ansonsten wird der nächste Befehl hinter dem LOOPE-Kommando ausgeführt.
Das LOOPE-Kommando wird normalerweise eingesetzt, um in Schleifen nach Werten ungleich Null zu suchen. Sie laden den Maximalwert der Schleife in das CX-Register und führen den Programmblock dann innerhalb eines LOOPE-Kommandos aus. Weil »Sprungziel« im Maschinenbefehl als 8-Bit-Entfernungswert codiert ist, darf es nur maximal 126 Byte vor oder 127 Byte nach dem LOOPE-Befehl liegen.
Beispiel:

```
MOV CX,100 ;lade 100 nach CX
MARKE MOV AL,ES:[CX] ;lade indiziertes Byte
LOOP MARKE ;und bleibe in Schleife, solange Byte = 0 und CX > 0
```

LOOPNE

»Loop while not equal«, führe Schleife aus, solange CX ungleich Null und das Zero-Flag gelöscht ist

Syntax: LOOPNE <Sprungziel>

Mit dem LOOPNE-Befehl veranlassen Sie, daß zunächst das CX-Register um eins dekrementiert wird. Falls es dann noch nicht den Wert Null besitzt, wird die Bearbeitung des Programms an der Marke »Sprungziel« fortgesetzt, sofern das Nullkennzeichen (»Zero-Flag«) gelöscht ist. Ansonsten wird der nächste Befehl hinter dem LOOPNE-Kommando ausgeführt.
Verwenden Sie das LOOPNE-Kommando, um in Schleifen nach Werten zu suchen, die gleich Null sind. Dazu laden Sie den Maximalwert der Schleife in das CX-Register und führen den Programmblock dann innerhalb eines LOOPNE-Kommandos aus. Weil »Sprungziel« im Maschinenbefehl als 8-Bit-Entfernungswert codiert ist, darf es nur maximal 126 Byte vor oder 127 Byte nach dem LOOPNE-Befehl liegen.
Beispiel:

```
MOV CX,100 ;lade 100 nach CX
MARKE MOV AL,ES:[CX] ;lade indiziertes Byte
LOOPNE MARKE ;und bleibe in der Schleife, falls Byte <> 0 und CX > 0
```

LOOPNZ

»Loop while not zero«

Syntax: LOOPNZ <Sprungziel>

Dieser Befehl ist identisch mit dem LOOPNE-Kommando und stellt nur eine alternative Schreibweise dar.

LOOPZ

»Loop while zero«

Syntax: LOOPZ <Sprungziel>

Dieser Befehl ist identisch mit dem LOOPE-Kommando und stellt nur eine alternative Schreibweise dar.

NOP

»No operation«, keine Ausführung

Syntax: NOP

Das NOP-Kommando bewirkt, daß der Prozessor den Befehl übergeht, ohne irgendwelche Aktionen auszuführen. Man verwendet den NOP-Befehl oft, um bestimmte Befehle in einem assemblierten Programm auszublenden, ohne es neu übersetzen zu

müssen. Sie überschreiben diese Befehle dann mit »NOPs«.
Beispiel:

```
NOP ;tue nichts, fahre beim nächsten Befehl fort
```

RET
»Return«, kehre aus einem Unterprogramm zurück

Syntax: RET

Mit »RET« beenden Sie ein mit dem CALL-Befehl aufgerufenes Unterprogramm. Das RET-Kommando weist den Prozessor an, den Wert für den Befehlszeiger vom Stapel zu holen und in das Registerpaar CS:IP (bei einer Far-Prozedur) oder IP (bei einer Near-Prozedur, siehe »CALL«) zu laden. Das Programm wird dadurch an dem auf CALL folgenden Befehl fortgesetzt.
Beispiel:

```
CALL UNTERP ;rufe Unterprogramm auf
...
UNTERP ... ;Start des Unterprogramms
...
RET ;kehre zurück
```

STI
»Set interrupt-enable-flag«, erlaube Interrupts

Syntax: STI

Dieser Befehl setzt das Interrupt-Kennzeichen (Flag). Mit dem Interrupt-Kennzeichen wird dem Prozessor mitgeteilt, ob maskierbare Unterbrechungen gesperrt oder zugelassen sind. Eine Blockade der Interrupts bewirkt der CLI-Befehl.
Beispiel:

```
CLI ;keine Interrupts mehr erlaubt
...
STI ;erlaube Interrupts
```

WAIT
»Wait«, warte

Syntax: WAIT

Der WAIT-Befehl schickt den Prozessor in einen Wartestatus, in dem keine weiteren Befehle ausgeführt werden. Dieser wird erst beendet, wenn die TEST-Leitung an dem Prozessor aktiv ist. Der WAIT-Befehl wird normalerweise verwendet, um den Prozessor zum Beispiel mit einem Coprozessor zu synchronisieren.
Beispiel:

```
WAIT ;warte auf TEST-Signal
```

(Martin Althaus/tr)

Assemblerreferenz: Speicher-, Port- und Stapelzugriff

Die wichtigsten Maschinenbefehle dienen dazu, den Inhalt von Speicherstellen in andere Speicherstellen oder in Register zu übertragen. Die folgende Übersicht enthält zusätzlich Kommandos, die den Prozessorstapel (»Stack«) verändern, und Befehle, die auf die Ein-/Ausgabe-Ports des PC zugreifen.

ENTER
»Create stack frame«, lege Parameterstruktur auf dem Stapel an

Syntax: ENTER <Größe>,<Ebene>

Dieser Befehl ist nur beim 80286- und 80386-Prozessor verfügbar und dient der Unterstützung höherer Programmiersprachen. Er legt eine Struktur für die Übergabe von Parametern an höhere Programmiersprachen an. Mit »Größe« bestimmen Sie, wie viele Bytes auf dem Stapel für die Prozedur reserviert werden sollen. Der Operand »Ebene« gibt die Anzahl der verschachtelten Ebenen der Prozedur an, also wie viele Parameterstrukturen auf dem Stack verschachtelt angelegt werden sollen.
Beispiel:

```
ENTER 10,3 ;zehn Bytes reservieren, Verschachtelungstiefe drei
```

IN
»Input«, hole Eingabe von einem Ein-/Ausgabe-Port

Syntax: IN <Zielregister>, <Port-Adresse>

Mit diesem Kommando wird ein Byte oder ein Wort von der angegebenen Port-Adresse in das »Zielregister« übertragen. Wenn Sie als Ziel das volle AX-Register angeben, dann liest der Prozessor ein Wort (2 Byte), ansonsten nur ein einzelnes Byte in die jeweilige Hälfte des Registers (also AL oder AH). Der 80386-Prozessor liest mit diesem Kommando auch Doppelworte (4 Byte), wenn Sie als Ziel das EAX-Register spezifizieren.

Die Port-Adresse muß einen gültigen Ein-/Ausgabe-Port darstellen. Sie kann unmittelbar als Konstante angegeben werden oder bereits im DX-Register stehen. Dann müssen Sie anstelle einer Port-Adresse natürlich das DX-Register einsetzen.

Beispiel:

```
MOV DX,03BFh ;lade Port-Adresse
IN AL,DX ;lese ein Byte nach AL
IN AH,DX ;und ein weiteres nach AH
```

LAHF
»Load register AH from flags«, lade das AH-Register mit den aktuellen Prozessor-Flags

Syntax: LAHF

Dieser Befehl überträgt die Bits aus dem Kennzeichenregister in das AH-Register. Dazu werden die Kennzeichenbits SF (»sign«), ZF (»zero«), AF (»auxiliary«), PF (»parity«) und CF (»carry«) in die Bits 7, 6, 4, 2 und 0 des AH-Registers übertragen. Die restlichen Bits des Registers (1, 3 und 5) bleiben ebenso wie die Prozessor-Flags selbst unverändert.

Beispiel:

```
LAHF ;übertrage Flags nach AH
TEST AH,1 ;Carry gesetzt?
JNZ GESETZT ;ja, dann springe zum Label »GESETZT«
```

LDS
»Load pointer using DS«, lade einen Zeiger auf eine Speicherstelle unter Verwendung des DS-Registers

Syntax: LDS <Zieloperand>,<Quelloperand>

Mit diesem Befehl übertragen Sie eine 32-Bit-Speicheradresse der Form »Segment:Offset« in ein Registerpaar. Dazu zeigt der Quelloperand (32 Bit) auf eine Adresse, ab der aufeinanderfolgend eine Offset- und eine Segmentadresse zu finden ist. Die Segmentadresse wird in das DS-Register geladen, die Offsetadresse in das mit »Zieloperand« angegebene Register. Der Zieloperand muß eines der 16-Bit-Register des Prozessors sein.
Beispiel:

```
ADRESSE DW 1234h,5678h ;Inhalt der Adresse: Segment 1234, Offset 5678hex
LDS SI,ADRESSE ;lade die Adresse 1234:5678hex nach DS:SI
```

LEA
»Load effective adress«, übertrage einen Offset in ein Register

Syntax: LEA <Zieloperand>,<Quelloperand>

Dieser Befehl überträgt den Offset des Quelloperanden (nicht seinen tatsächlichen Inhalt) in den Zieloperanden. Der Quelloperand muß eine Speicheradresse und der Zieloperand eines der 16-Bit-Register sein. Im Gegensatz zum MOV-Befehl darf der Quelloperand über das SI-Register indiziert sein, »MOV BX,OFFSET TABELLE[SI]« wäre ungültig, »LEA BX, TABELLE[SI]« ist dagegen zulässig.
Beispiel:

```
LEA DX,MSG ;lade Adresse einer Meldung
MOV AH,9 ;lade Funktionsnummer 9
INT 21h ;gebe Meldung aus
```

LEAVE
»Resolve stack frame«, lösche eine mit dem ENTER-Befehl angelegte Struktur vom Stapel

Syntax: LEAVE

Dieses Kommando ist nur beim 80286- und 80386-Prozessor verfügbar. Es löst eine mit dem ENTER-Befehl aufgebaute Parameterstruktur auf dem Stapel wieder auf. Der alte Zustand des Stapels vor Aufruf der Prozedur wird damit wiederhergestellt.
Beispiel:

```
LEAVE ;löse eine Datenstruktur auf
```

LES
»Load pointer using ES«, lade einen Zeiger auf eine Speicherstelle unter Verwendung des ES-Registers

Syntax: LES <Zieloperand>,<Quelloperand>

Der LES-Befehl arbeitet analog zum LDS-Kommando, nur wird hier die Segmentadresse des Quelloperanden in das ES-Register geladen.
Beispiel:

```
ADRESSE DW 1234h,5678h ;Inhalt der Adresse: Segment 1234, Offset 5678hex
LES SI,ADRESSE ;lade die Adresse
1234:5678hex nach ES:SI
```

MOV
»Move byte or word«, übertrage ein Byte oder ein Wort

Syntax: MOV <Zieloperand>,<Quelloperand>

Der MOV-Befehl kopiert ein Byte oder ein Wort aus dem Quelloperanden in den Zieloperanden. Beim 80386-Prozessor kann dieser Befehl auch Doppelworte übertragen.
Für die Kombination des Zieloperanden und des Quelloperanden sind folgende Adressierungsarten zulässig:

```
Register / Register
Speicheradresse / AX-Register
AX-Register / Speicheradresse
Speicheradresse / Register
Register / Speicheradresse
Speicheradresse / Konstante
Register / Konstante
Segmentregister / 16-Bit-Register
Segmentregister / 16-Bit-Speicheradresse
16-Bit-Register / Segmentregister
Speicheradresse / Segmentregister
```

aber nicht:

```
Speicheradresse / Speicheradresse
```

Beispiel:

```
MOV CX,100 ;lade 100 nach CX
XOR AH,AH ;lösche AH
MARKE MOV SPEICHER[CX],AH ;lösche das Byte an der Position »SPEICHER + CX«
LOOP MARKE ;bilde Schleife (100mal)
```

OUT
»Output«, übertrage einen Wert auf einen Ein-/Ausgabe-Port

Syntax: OUT <Port-Adresse>, <Register>

Mit diesem Kommando wird ein Byte oder ein Wort aus einem Register an eine Port-Adresse übertragen. Wenn Sie für »Register« das volle AX-Register angeben, dann wird ein Wort (2 Byte) ausgegeben, ansonsten nur ein Byte aus der jeweiligen Hälfte des Registers (also AL oder AH). Der 80386-Prozessor kann mit diesem Befehl auch Doppelworte ausgeben, wenn das EAX-Register als Operand folgt.
Die Port-Adresse muß auf einen gültigen Ein-/Ausgabe-Port zeigen. Sie kann unmittelbar angegeben oder zuvor in das DX-Register geladen werden. Anstelle einer Port-Adresse muß in diesem Fall natürlich »DX« stehen.
Beispiel:

```
MOV DX,03BFh ;lade Port-Adresse
XOR AH,AH ;lösche AH-Register
OUT AH,DX ;gebe Byte an den Port DX aus
OUT AH,DX ;und gebe ein weiteres aus
```

POP
»Pop a word from the stack«, hole ein Speicherwort vom Prozessorstapel

Syntax: POP <Zieloperand>

Der POP-Befehl überträgt das zuerst auf dem Stapel liegende Wort in den Zieloperanden und aktualisiert den Stapelzeiger. Der Zieloperand darf ein 16-Bit-Register (ausgenommen CS) oder eine 16-Bit-Speicheradresse sein. Beim 80386-Prozessor können mit dem POP-Kommando auch Doppelworte vom Stapel geholt werden, wenn der Zieloperand im 32-Bit-Format vorliegt.
Man benutzt diesen Befehl zusammen mit dem PUSH-Kommando zur vorübergehenden Sicherung von Daten, zum Beispiel in Unterprogrammen. Zu Beginn eines Programms werden die Daten mit dem PUSH-Befehl auf dem Stapel gesichert und am Ende des Programms mit dem POP-Kommando wieder vom Stapel geholt.
Beispiel:

```
POP AX ;hole ein Wort vom Stapel und lege es im AX-Register ab
```

POPA
»Pop all registers from the stack«, hole den kompletten Registersatz vom Stapel

Syntax: POPA

Dieser Befehl dient dazu, alle 16-Bit-Register mit einem einzigen Kommando vom Stapel zu holen. POPA ist nur beim 80286- und 80386-Prozessor verfügbar. Der Rechner lädt die Register in der Reihenfolge DI, SI, BP, SP, BX, DX, CX und AX und aktualisiert den Stapelzeiger. Das POPA-Kommando ist das Gegenstück zum PUSHA-Befehl.
Beispiel:

```
POPA ;hole Register vom Stack
```

POPAD
»Pop all double registers from the stack«, hole alle 32-Bit-Register des 80386-Prozessors vom Stapel

Syntax: POPAD

Die POPAD-Anweisung holt alle 32-Bit-Register vom Stapel. Das Kommando ist nur beim 80386-Prozessor verfügbar, da nur dieser über interne 32-Bit-Register verfügt. Der Prozessor lädt die Register in der Reihenfolge EDI, ESI, EBP, ESP, EBX, EDX, ECX und EAX und aktualisiert den Stapelzeiger. Das POPAD-Kommando ist das Gegenstück zum PUSHAD-Befehl.
Beispiel:

```
POPAD ;hole Register vom Stack
```

POPF
»Pop flags from stack«, hole die Prozessor-Flags vom Stapel

Syntax: POPF

POPF überträgt das oberste Element des Stapels in das Kennzeichenregister des Prozessors und aktualisiert den Stapelzeiger.
Man verwendet diesen Befehl, um die mit dem PUSHF-Kommando auf den Stapel geretteten Informationen über besondere Prozessorzustände oder über das Ergebnis einer Operation wieder in das Kennzeichenregister zu übertragen.
Beispiel:

```
POPF ;hole Flags vom Stapel
```

PUSH
»Push word onto stack«, lege ein Wort oben auf den Prozessorstapel

Syntax: PUSH <Quelloperand>

Der Inhalt des Quelloperanden wird auf dem Stapel abgelegt und der Stapelzeiger aktualisiert. Der Quelloperand darf ein beliebiges 16-Bit-Register oder eine Speicheradresse im 16-Bit-Format sein. Der 80386-Prozessor erkennt den Quelloperanden auch im 32-Bit-Format.
Man benutzt diesen Befehl zur temporären Sicherung von Daten, zum Beispiel in Unterprogrammen. Zu Beginn eines Programms werden die Daten mit dem PUSH-Befehl auf dem Stapel gesichert und am Ende des Programms mit dem POP-Kommando wieder vom Stapel geholt.
Beispiel:

```
PUSH AX ;rette AX,
PUSH BX ;BX,
PUSH CX ;und CX auf den Stapel
```

PUSHA
»Push all registers onto stack«, schiebe alle Register auf den Prozessorstapel

Syntax: PUSHA

Dieser Befehl dient dazu, alle 16-Bit-Register des 80286- oder 80386-Prozessors mit einem Kommando auf den Stapel zu schieben. Der Prozessor speichert die Register in der Reihenfolge DI, SI, BP, SP, BX, DX, CX und AX und aktualisiert den Stapelzeiger. Das PUSHA-Kommando ist das Gegenstück zum POPA-Befehl.
Beispiel:

```
PUSHA ;schiebe Register auf den Stapel
```

PUSHAD
»Push all double registers onto stack«, lege alle 32-Bit-Register des 80386 auf dem Stapel ab

Syntax: PUSHAD

Dieser Befehl dient dazu, alle 32-Bit-Register des 80386-Prozessors mit einem Kommando auf den Stapel zu schieben. Der Prozessor speichert die Register in der Reihenfolge EDI, ESI, EBP, ESP, EBX, EDX, ECX und EAX und aktualisiert den Stapelzeiger.
Das PUSHAD-Kommando ist das Gegenstück zum POPAD-Befehl.
Beispiel:

```
PUSHAD ;hole Register vom Stack
```

PUSHF
»Push flags onto the stack«, schiebe die Prozessor-Flags auf den Stapel

Syntax: PUSHF

PUSHF überträgt das Kennzeichenregister des Prozessors auf den Stapel und aktualisiert den Stapelzeiger.
Man benutzt den Befehl, um besondere Zustände und Ergebnisse einer Operation zur späteren Verwendung zwischenzuspeichern. Mit dem POPF-Befehl übertragen Sie das auf dem Stapel liegende Kennzeichenregister später wieder in den Prozessor.
Beispiel:

```
PUSHF ;lege die Prozessor-Flags auf den
Stapel
```

SAHF
»Store register AH into flags«, übertrage den Inhalt des AH-Registers in das Flag-Register

Syntax: SAHF

Der SAHF-Befehl kopiert die mit den Kennzeichenbits korrespondierenden Bits des AH-Registers in das Kennzeichenregister. Dazu werden die Bits 7, 6, 4, 2 und 0 des AH-Registers in die Kennzeichen SF, ZF, AF, PF und CF des Kennzeichenregisters übertragen.
Beispiel:

```
MOV AH,1 ;Bit 0 gesetzt
SAHF ;übertrage AH nach Flags - setze das
Carry-Flag
```

XCHG
»Exchange«, vertausche zwei Werte

Syntax: XCHG <Quelloperand>,<Zieloperand>

Dieses Kommando vertauscht den Inhalt des Ziel- und des Quelloperanden miteinander. Die Operanden können Bytes oder Wörter sein, beim 80386-Prozessor auch Doppelwörter. Der Quelloperand darf entweder eine Speicher- oder eine Registerangabe sein, der Zieloperand muß ein Register sein. Es können keine Inhalte von Speicherstellen direkt miteinander ausgetauscht werden.
Beispiel:

```
XCHG AX,BX ;tausche AX und BX
XCHG WERT,AX ;tausche den Inhalt von
»WERT« und AX
```

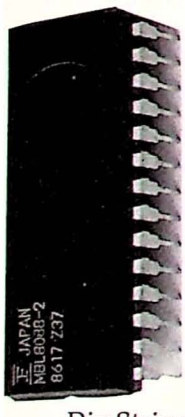
XLAT
»Translate«, lade das AL-Register mit einem Tabellenwert

Syntax: XLAT <Quelltabelle>

Dieser Befehl ersetzt den Inhalt des AL-Registers durch ein Byte aus der Quelltabelle. Es wird davon ausgegangen, daß das BX-Register (»Segment:Offset«) auf den Anfang einer Tabelle zeigt und AH einen Index in dieser Tabelle darstellt. Dann wird AL durch das Byte ersetzt, das relativ zum Tabellenanfang an der Stelle AH steht. Der XLAT-Befehl arbeitet damit genauso wie ein »MOV AL,DS:[BX + AL]«, das in dieser Form jedoch nicht zulässig wäre.
Streng genommen ist der Parameter »Quelltabelle« für die Codierung des XLAT-Befehls nicht notwendig, weil der Prozessor davon ausgeht, daß BX und AL die Tabelle adressieren. Sie dürfen daher bei vielen Assemblern auch einfach nur »XLAT« ohne Parameter schreiben.
Beispiel:

```
LEA BX,TABELLE ;lade Adresse der Tabelle
MOV AH,6 ;das sechste Byte soll gelesen
werden
XLAT TABELLE ;lade das Byte nach AL
```

(Martin Althaus/tr)



Assemblerreferenz: Zeichenketten

Die Intel-Prozessoren besitzen einige leistungsfähige Befehle, die speziell die Verarbeitung von Zeichenfolgen unterstützen. Diese Übersicht enthält alle Kommandos zum Kopieren, Vergleichen, Durchsuchen, Ausgeben und Einlesen von Zeichenketten.

Die String-Befehle (englisch »string«: Zeichenkette) dienen der Verarbeitung ganzer Folgen von Bytes oder Wörtern. Jeder auf diese Weise zusammenhängende Block von Informationen, seien es nun Binärzahlen, BCD-Zahlen, ASCII-Zeichen und so weiter, wird als String bezeichnet. Die String-Befehle beim 80386-Prozessor beziehen sich auch auf Doppelwörter.

Die String-Befehle kopieren, vergleichen, durchsuchen, speichern und lesen Datenblöcke. Dabei geht der Prozessor immer davon aus, daß sich der Quell-String für die jeweilige Operation im aktuellen Datensegment befindet, das vom SI-Register adressiert wird, während sich der Ziel-String im Extrasegment befindet, das vom DI-Register adressiert wird. Soll der Prozessor beispielsweise zwei Strings vergleichen, dann setzen Sie zuerst diese Zeiger auf die Strings und führen dann den Vergleich durch.
Sie definieren Strings in Ihren Assemblerprogrammen durch entsprechende Anweisungen im Datensegment, zum Beispiel:

```
quellstring db 256 dup(0)
zielstring dw 1024 dup(?)
string_eins dd 256 dup(1)
```

Je nachdem, ob Sie die Strings als Byte-, Wort- oder Doppelwortblöcke

definieren, bestimmen Sie damit auch die Anwendung des späteren String-Befehls. Sie können beispielsweise einen mit »db« deklarierten String nicht mit einem Befehl bearbeiten, der sich auf Wörter von zwei Byte Länge bezieht.
Die meisten Befehle beziehen sich dabei nur auf ein einzelnes Byte, Wort oder Doppelwort. Um sie auf einen kompletten String anzuwenden, setzen Sie den Befehlen sogenannte »Präfixe« voran. Diese Maschinenbefehle zeigen an, daß ein bestimmter Befehl, abhängig von bestimmten Bedingungen, mehrmals hintereinander ausgeführt werden soll.
Obwohl diese Wiederholungspräfixe eigentlich Kommandos zur Ablaufsteuerung sind, finden Sie sie in dieser Referenzübersicht bei den String-Befehlen, weil sie sich nur in Zusammenhang mit String-Befehlen sinnvoll einsetzen lassen.
Damit der Prozessor einen kompletten String verarbeitet, vermindert oder erhöht er nach der Ausführung eines String-Befehls die Zeiger auf den Quell-String und den Ziel-String (also die Register SI und DI) bei byteorientierten Befehlen um 1 und bei wortorientierten Befehlen um 2. Bei den nur beim 80386-Prozessor verfügbaren String-Befehlen für Doppelworte werden die Zeiger entsprechend um 4 erhöht oder vermindert. Ob sie erhöht oder vermindert werden, hängt dabei vom Zustand des Richtungskennzeichens (»Direction Flag«) ab. Ist es gesetzt, werden die Zeiger erhöht, ist es gelöscht, werden sie vermindert.

CLD

»Clear direction flag«, lösche das Richtungskennzeichen

Syntax: CLD

Über den Zustand des Richtungskennzeichens (»Direction Flag«) teilen Sie dem Prozessor mit, ob er sich bei mehrfacher Anwendung von String-Befehlen von den niedrigen Speicheradressen zu den höheren vorarbeitet, also den String »vorwärts« liest oder umgekehrt. Ein gesetztes Richtungskennzeichen bedeutet, daß die String-Zeiger SI und DI inkrementiert, und ein gelöschtes Richtungskennzeichen, daß die String-Zeiger dekrementiert werden sollen.
Beispiel:

```
CLD ;lösche Richtungskennzeichen
```

CMPS

»Compare string«, vergleiche Teile zweier Zeichenketten

Syntax: CMPS <Ziel-String>,<Quell-String>

Der CMPS-Befehl vergleicht ein Zeichen aus einem String mit einem anderen Zeichen aus einem anderen String. Dabei wird der Quell-String (adressiert durch DS:SI) vom Ziel-String (adressiert durch ES:DI) subtrahiert. Der Prozessor setzt die Kennzeichenbits dann entsprechend dem Ergebnis der Subtraktion. Die beiden Operanden »Ziel-String« und »Quell-String« dienen dem Assembler nur dazu, um festzustellen, ob es sich bei dem String um einen Byte-String oder einen Wort-String handelt. Beim 80386 kann das Kommando auch Strings verarbeiten, die aus Doppelworten bestehen.
Beachten Sie jedoch, daß das CMPS-Kommando selbst keinen gültigen Maschinenbefehl darstellt. Der Assembler übersetzt den CMPS-Befehl je nach Typ der Operanden in einen CMPSB-, einen CMPSW- oder einen CMPSD-Befehl.
Nach der Ausführung des Befehls werden die Index-Register SI und DI je nach Zustand des Richtungskennzeichens um die Werte 1, 2 oder 4 (je nach Typ) entweder erhöht oder erniedrigt.
Beispiel:

```
MOV SI,OFFSET STRING1 ;erster String
MOV DI,0 ;zweiter String in ES
MOV CX,100 ;String ist 100 Zeichen lang
REPE CMPS STRING1,STRING1 ;vergleiche
die Strings
```

CMPSB

»Compare string byte«, vergleiche ein Byte von zwei Strings

Syntax: CMPSB

Dieses Kommando ist einer der drei Befehle, in die ein CMPS-Befehl vom Assembler übersetzt wird. Im Gegensatz zum CMPS-Kommando benutzt »CMPSB« keine Operanden, weil der Prozessor implizit davon ausgeht, daß der Quell-String durch DS:SI und der Ziel-String durch ES:DI adressiert ist.
Das aktuelle Byte des Ziel-Strings wird vom Byte des Quell-Strings abgezogen. Je nach Ergebnis dieser Subtraktion setzt der Rechner entsprechend die Flags. Danach erhöht oder erniedrigt er abhängig vom Zustand des Richtungskennzeichens die Zeiger DI und SI um den Wert 1.
Beispiel:

```
MOV SI,OFFSET STRING1 ;lade Adresse von
erstem String
MOV DI,0 ;Adressiere erstes Byte im ES-
Register
CMPSB ;vergleiche String-Bytes
```

CMPSW

»Compare string word«, vergleiche Wörter von zwei Strings

Syntax: CMPSW

Diese Anweisung ist eine der drei Befehle, in die ein CMPS-Befehl vom Assembler übersetzt wird. Im Gegensatz zum CMPS-Kommando benutzt »CMPSW« keine Operanden, weil der Prozessor implizit davon ausgeht, daß der Quell-String durch DS:SI und der Ziel-String durch ES:DI adressiert ist.
Das aktuelle Wort (zwei Byte) des Ziel-Strings wird vom Wort des Quell-Strings abgezogen. Je nach Ergebnis dieser Subtraktion setzt der Rechner entsprechend die Flags. Danach erhöht oder erniedrigt er die Zeiger DI und SI um den Wert 2, abhängig vom Zustand des Richtungskennzeichens.
Beispiel:

```
MOV SI,OFFSET STRING1 ;lade Adresse von
erstem String
MOV DI,0 ;Adressiere erstes Wort im ES-
Register
CMPSW ;vergleiche String-Bytes
```

CMPSD

»Compare string doubleword«, vergleiche Doppelwörter von zwei Strings (nur 80386)

Syntax: CMPSD

Dieses Kommando ist einer der drei Befehle, in die ein CMPS-Befehl vom Assembler übersetzt wird; es ist nur beim 80386-Prozessor verfügbar. Im Gegensatz zum CMPS-Kommando benutzt »CMPSD« keine Operanden, weil der Prozessor davon ausgeht, daß der Quell-String durch DS:SI und der Ziel-String durch ES:DI adressiert ist.
Das aktuelle Doppelwort (vier Byte) des Ziel-Strings wird vom Doppelwort des Quell-Strings abgezogen. Je nach Ergebnis dieser Subtraktion setzt der Rechner entsprechend die Flags. Danach erhöht oder erniedrigt er abhängig vom Zustand des Richtungskennzeichens die Zeiger DI und SI um den Wert 4.
Beispiel:

```
MOV SI,OFFSET STRING1 ;lade Adresse von
erstem String
MOV DI,0 ;Adressiere erstes Wort im ES-
Register
CMPSD ;vergleiche String-Bytes
```

INS

»Input of a string«, hole Teil eines Strings von einem Port (nur 80286 und 80386)

Syntax: INS <Ziel-String>,DX

Dieses Kommando ist nur auf dem 80286- und dem 80386-Prozessor verfügbar. Es überträgt Werte des über das DX-Register adressierten Ports in den Ziel-String. Der Assembler ermittelt den Datentyp des Ziel-Strings und liest ein Byte, ein Wort oder ein Doppelwort (nur beim 80386) vom

Port DX aus und speichert es an der mit ES:DI adressierten Stelle im String. Nach Ausführung des Befehls wird das DI-Register um den Wert 1, 2 oder 4 (Byte, Wort, Doppelwort) je nach Zustand des Richtungskennzeichens erhöht oder vermindert
Der INS-Befehl selbst ist kein gültiger Maschinenbefehl. Der Ziel-String wird nur verwendet, um den Datentyp des Strings zu ermitteln. Dann übersetzt der Assembler »INS« in eines der Kommandos »INSB«, »INSW« oder »INSD« (letzteres nur beim 80386).
Beispiel:

```
;ES muß schon auf das Extrasegment zeigen
MOV DX,03BFh ;lade Portadresse 03BFhex
LEA DI,ES:STRING ;lade Adresse des
Strings
MOV CX,10 ;Zähler: 10 Werte lesen
REP INS STRING,DX ;lies Werte aus
```

INSB
»Input of a string byte«, kopiere ein Byte von einem Port in einen String (nur 80286 und 80386)

Syntax: INSB

Dieser Befehl ist einer der drei Befehle, in die der Assembler den INS-Befehl übersetzt. Mit »INSB« wird ein Byte von dem durch DX adressierten Port gelesen und in dem durch ES:DI adressierten Byte-String gespeichert.
Beispiel:

```
;ES muß bereits auf das Extrasegment
zeigen
LEA ES:STRING1 ;lade Adresse des Strings
MOV DX,03BFh ;lade Portadresse
INSB ;kopiere ein Byte in den String
```

INSW
»Input of a string word«, kopiere ein Wort von einem Port in einen String (nur 80286 und 80386)

Syntax: INSW

Dieser Befehl ist einer der drei Befehle, in die der Assembler den INS-Befehl übersetzt. Mit »INSW« wird ein Wort von dem durch DX adressierten Port eingelesen und in dem durch ES:DI adressierten Wort-String gespeichert.
Beispiel:

```
;ES muß bereits auf das Extrasegment
zeigen
LEA ES:STRING1 ;lade Adresse des Strings
MOV DX,03BFh ;lade Portadresse
INSW ;kopiere ein Wort in den String
```

INSD
»Input of a string double word«, kopiere ein Doppelwort von einem Port in einen String (nur 80386)

Syntax: INSD

Dieser Befehl ist einer der drei Befehle, in die der Assembler den INS-Befehl übersetzt. Mit »INSD« wird ein Doppelwort von dem durch DX adressierten Port eingelesen und in dem durch ES:DI adressierten Doppelwort-String gespeichert. Der INSD-Befehl ist nur beim 80386-Prozessor einsetzbar, da nur dieser Strings, bestehend aus Doppelwörtern, verarbeiten kann.
Beispiel:

```
;ES muß bereits auf das Extrasegment
zeigen
LEA ES:STRING1 ;lade Adresse des Strings
MOV DX,03BFh ;lade Portadresse
INSD ;kopiere ein Doppelwort in den
String
```

LODS
»Load string«, kopiere den Teil eines Strings in das AX-Register

Syntax: LODS <Quell-String>

Dieses Kommando lädt ein Element des mit DS:SI adressierten Quell-Strings in den Akkumulator. Der Operand »Quell-String« dient dabei dem Assembler lediglich zur Feststellung, ob es sich bei dem Quell-String um einen Byte-, Wort- oder Doppelwort-String handelt. Im ersten Fall überträgt der Prozessor das gerade durch SI adressierte Byte nach AL, im zweiten Fall das durch SI adressierte Wort nach AX, im dritten Fall das durch ES:DI adressierte Doppelwort nach EAX. Nach der Übertragung des Bytes oder des Wortes wird das Register SI je nach Datentyp und je nach Zustand des Richtungskennzeichens entweder um 1, 2 oder 4 erhöht oder vermindert.
Beachten Sie, daß »LODS« selbst keinen gültigen Maschinenbefehl darstellt. Der Assembler bestimmt anhand des Quell-Strings den Datentyp des Strings und übersetzt das Kommando entweder in den LODSB-, LODSW- oder LODSD-Befehl.
»LODS« ist dann nützlich, wenn Sie beispielsweise zwei Strings zuerst auf Gleichheit überprüfen und das erste ungleiche Element dann in den Akkumulator laden möchten.
Beispiel:

```
LEA SI,QUELLSTRING ;lade Adresse des
Strings
LODS QUELLSTRING ;und lade erstes Zeichen
MOV ERST,AX ;speichere es
LODS QUELLSTRING ;und lade zweites Zei-
chen
```

LODSB
»Load string byte«, kopiere ein Byte eines Strings in das AL-Register

Syntax: LODSB

Dieses Kommando ist einer der drei Befehle, in die ein LODS-Befehl vom Assembler übersetzt wird. Das LODSB-Kommando geht davon aus, daß SI auf einen Byte-String in dem durch DS adressierten Datensegment zeigt. Durch »LODSB« wird das aktuell durch SI indizierte Byte-Element des Strings in das AL-Register geladen. Danach erhöht oder vermindert der Prozessor den Zeiger SI entsprechend dem Zustand des Richtungskennzeichens um 1.
Beispiel:

```
LEA SI,QUELLSTRING ;lade Adresse Quell-
String
LODSB ;lade erstes Byte
MOV ERST,AL ;speichere es
LODSB ;und lade zweites Byte
```

LODSW
»Load string word«, kopiere ein Wort eines Strings in das AX-Register

Syntax: LODSW

Dieses Kommando ist einer der drei Befehle, in die ein LODS-Befehl vom Assembler übersetzt wird. Das LODSW-Kommando geht davon aus, daß SI auf einen Wort-String in dem durch DS adressierten Datensegment zeigt. Durch LODSW wird das aktuell durch SI indizierte Wortelement des Strings in das AX-Register geladen. Danach erhöht oder vermindert der Prozessor den Zeiger SI entsprechend dem Zustand des Richtungskennzeichens um 2.
Beispiel:

```
LEA SI,QUELLSTRING ;lade Adresse Quell-
string
LODSW ;lade erstes Wort
MOV ERST,AX ;speichere es
LODSW ;und lade zweites Wort
```

LODSD
»Load string double word«, kopiere ein Doppelwort eines Strings in das EAX-Register (nur 80386)

Syntax: LODSD

Dieses nur beim 80386-Prozessor verfügbare Kommando ist einer der drei Befehle, in die ein LODS-Befehl vom Assembler übersetzt wird. Das LODSD-Kommando geht davon aus, daß SI auf einen Doppelwort-String in dem durch ES adressierten Datensegment zeigt. Durch »LODSD« wird das aktuell durch SI indizierte Doppelwortelement des Strings in das EAX-Register geladen. Danach erhöht oder vermindert der Prozessor den Zeiger ES:SI entsprechend dem Zustand des Richtungskennzeichens um 4.
Beispiel:

```
LEA SI,QUELLSTRING ;lade Adresse Quell-
string
LODSD ;lade erstes Doppelwort
MOV ERST,EAX ;speichere es
LODSD ;und lade zweites Doppelwort
```

MOVS
»Move string«, kopiere ein Element eines Strings

Syntax: MOVS <Ziel-String>,
<Quell-String>

Dieser Befehl lädt ein Element einer Zeichenkette und kopiert es in eine andere Zeichenkette. Dazu wird das mit DS:SI adressierte Element eines Quell-Strings in das mit ES:DI adressierte Element eines Ziel-Strings übertragen. Der Assembler verwendet die Operanden »Ziel-String« und »Quell-String« nur zur Ermittlung des Datentyps der Strings. Handelt es sich um einen Wort-String, so wird ein komplettes Wort übertragen, handelt es sich hingegen um einen Byte-String, wird ein Byte übertragen. Bei einem Doppelwort-String schließlich (nur 80386), kopiert der Prozessor ein ganzes Doppelwort.
Nach der Ausführung des Befehls werden die Zeiger für SI und DI je nach Zustand des Richtungskennzeichens und je nach String-Typ um die Werte 1, 2 oder 4 erhöht oder vermindert.
Das MOVS-Kommando selbst stellt keinen gültigen Maschinenbefehl dar. Der Assembler verwendet die Angabe der Parameter nur, um den Datentyp der Strings zu ermitteln und entweder den MOVSB-, MOVSW- oder MOVSD-Befehl in den Programmcode einzusetzen.
Beispiel:

```
LEA SI,QUELLSTRING ;lade Adresse des
Quell-Strings
LEA DI,ES:ZIELSTRING ;lade Adresse des
Ziel-Strings
MOVS QUELLSTRING,ZIELSTRING ;und über-
trage ein Zeichen
```

MOVSB

»Move string byte«, kopiere ein Byte von einem String in einen anderen String

```
Syntax: MOVSB
```

Dieses Kommando ist einer der drei Maschinenbefehle, in die der Assembler das MOVS-Kommando übersetzt. »MOVSB« überträgt das mit DS:SI adressierte Byte eines Strings in das mit ES:DI adressierte Byte eines anderen Strings. Danach werden die Register SI und DI je nach Zustand des Richtungskennzeichens um 1 erhöht oder vermindert.
Beispiel:

```
LEA SI,QUELLSTRING ;lade Adresse des
Quell-Strings
LEA DI,ZIELSTRING ;lade Adresse des Ziel-
Strings
MOVSB ;übertrage erstes...
MOVSB ;... und zweites Byte
```

MOVSW

»Move string word«, kopiere ein Wort von einem String in einen anderen String

```
Syntax: MOVSW
```

Dieses Kommando ist einer der drei Maschinenbefehle, in die der Assembler das MOVS-Kommando übersetzt. »MOVSW« überträgt das mit DS:SI adressierte Wort eines Strings in das mit ES:DI adressierte Wort eines anderen Strings. Danach werden die Register SI und DI je nach Zustand des Richtungskennzeichens um den Wert 2 erhöht oder vermindert.
Beispiel:

```
LEA SI,QUELLSTRING ;lade Adresse des
Quell-Strings
LEA DI,ZIELSTRING ;lade Adresse des Ziel-
Strings
MOVSW ;übertrage erstes...
MOVSW ;...und zweites Wort
```

MOVSD

»Move string double word«, kopiere ein Doppelwort aus einem String in einen anderen String (nur 80386)

```
Syntax: MOVSD
```

Dieses Kommando ist einer der drei Maschinenbefehle, in die der Assembler das MOVS-Kommando übersetzt. »MOVSD« überträgt das mit DS:SI adressierte Doppelwort eines Strings in das mit ES:DI adressierte Doppelwort eines anderen Strings. Danach werden die Register SI und DI je nach Zustand des Richtungskennzeichens um den Wert 4 erhöht oder vermindert. MOVSD ist nur auf dem 80386 einsetzbar, da nur dieser Prozessor Zeichenketten, die aus Doppelworten bestehen, verarbeitet.
Beispiel:

```
LEA SI,QUELLSTRING ;lade Adresse des
Quell-Strings
LEA DI,ZIELSTRING ;lade Adresse des Ziel-
Strings
MOVSD ;übertrage erstes...
MOVSD ;...und zweites Doppelwort
```

OUTS

»Output of a string«, übertrage den Teil eines Strings an einen Port (nur 80286 und 80386)

```
Syntax: OUTS <Quell-String>
```

Dieses Kommando ist nur auf den 80286- und 80386-Prozessoren verfügbar. Es überträgt Werte aus dem Quell-String auf den über DX adressierten Port. Der Assembler ermittelt den Datentyp des Quell-Strings, liest ein Byte, Wort oder Doppelwort (nur 80386) von der Position ES:DI und gibt es auf die im DX-Register gefundene Portadresse aus. Je nach Typ des Quell-Strings und je nach Zustand des Richtungskennzeichens erhöht oder erniedrigt der Prozessor nach der Operation das DI-Register um die Werte 1, 2 oder 4.
Beachten Sie, daß der OUTS-Befehl selbst keinen gültigen Maschinenbefehl darstellt. Der Quell-String wird vom Assembler nur dazu verwendet, um den Datentyp des Strings zu ermitteln. Dann wird »OUTS« in eines der Kommandos »OUTSB«, »OUTSW« oder »«OUTSD« (nur 80386) übersetzt.
Beispiel:

```
;ES muß bereits auf das Extrasegment
zeigen
MOV DX,03BFh ;lade Portadresse 03BFhex
LEA DI,ES:STRING ;lade Adresse des
Strings
MOV CX,10 ;Zähler: 10 Werte schreiben
REP OUTS DX,STRING ;kopiere Zeichen
```

OUTSB

»Output of a string byte«, gib ein Byte eines Strings auf einen Port aus (nur 80286 und 80386)

```
Syntax: OUTSB
```

Dieser Befehl ist einer der drei Befehle, in die der Assembler den OUTS-Befehl übersetzt. Mit »OUTSB« wird ein Byte des durch ES:DI adressierten Strings auf dem über DX adressierte Port ausgegeben. Der Befehl ist nur auf den 80286- und 80386-Prozessoren verfügbar.
Beispiel:

```
;ES muß bereits auf das Extrasegment zeigen
LEA ES:STRING ;lade Adresse des Strings
MOV DX,03BFh ;lade Portadresse
OUTSB ;schreibe String-Byte
```

OUTSW

»Output of a string word«, gib ein Wort eines Strings auf einem Port aus (nur 80286 und 80386)

```
Syntax: OUTSW
```

Dieser Befehl ist einer der drei Befehle, in die der Assembler den OUTS-Befehl übersetzt. Mit »OUTSW« wird ein Wort des durch ES:DI adressierten Strings auf dem durch DX adressierten Port ausgegeben. Der Befehl ist nur auf den 80286- und 80386-Prozessoren verfügbar.
Beispiel:

```
;ES muß bereits auf das Extrasegment
zeigen
LEA ES:STRING ;lade Adresse des Strings
MOV DX,03BFh ;lade Portadresse
OUTSW ;schreibe String-Wort
```

OUTSD

»Output of a string double word«, gibt ein Doppelwort eines Strings auf einem Port aus (nur 80386)

```
Syntax: OUTSD
```

Dieser Befehl ist einer der drei Befehle, in die der Assembler den OUTS-Befehl übersetzt. Mit »OUTSD« wird ein Doppelwort des durch ES:DI adressierten Strings auf dem durch DX adressierten Port ausgegeben. Der Befehl ist nur beim 80386-Prozessor verfügbar, da nur dieser Strings aus Doppelwörtern verarbeitet.
Beispiel:

```
;ES muß bereits auf das Extrasegment
zeigen
LEA ES:STRING ;lade Adresse des Strings
MOV DX,03BFh ;lade Portadresse
OUTSD ;schreibe String-Doppelwort
```

REP
»Repeat«, wiederhole eine String-Anweisung

Syntax: REP <String-Befehl> <Parameter>

Der REP-Befehl ist kein eigenständiger Maschinenbefehl, sondern ein sogenanntes »Wiederholungspräfix«. Sie setzen »REP« vor einen beliebigen String-Befehl, der je nach Syntax einige Parameter hat. Der Befehl hinter dem REP-Präfix wird dann so lange wiederholt, bis das CX-Register den Wert 0 besitzt. Wenn der String-Befehl eine der beiden Vergleichsoperationen »CMPS« oder »SCAS« ist, so bricht der Prozessor die Schleife auch dann ab, wenn das Zero-Flag gelöscht wird, er also eine Differenz bei den zu vergleichenden Strings findet. Nach jeder Ausführung des jeweiligen String-Befehls zählt der Prozessor das CX-Register um 1 herab.
Man benutzt den Befehl in Schleifen, die ganze Zeichenketten verarbeiten sollen, zum Beispiel beim MOVS-Kommando. Bei diesen Befehlen tragen Sie zuerst in das CX-Register die Länge des Strings ein und verarbeiten dann mit dem REP-Präfix den kompletten String.
Beispiel:

```
MOV CX,100 ;der String ist 100 Worte lang
LEA SI,STRING1 ;lade Adresse des ersten Strings
LEA DI,ES:STRING2 ;lade Adresse des zweiten Strings
REP MOVSW ;übertrage einzelnes String-Wort in einer Schleife
```

REPE
»Repeat while equal«, wiederhole eine String-Anweisung, solange das Zero-Flag gesetzt ist

Syntax: REPE <String-Befehl> <Parameter>

Der REPE-Befehl ist identisch mit dem REP-Kommando und stellt lediglich eine alternative Schreibweise dar.

REPZ
»Repeat while zero«, wiederhole eine String-Anweisung, solange das Zero-Flag gesetzt ist

Syntax: REPZ <String-Befehl> <Parameter>

Der REPZ-Befehl ist identisch mit dem REP-Kommando und stellt lediglich eine alternative Schreibweise dar.

REPNE
»Repeat while not equal«, wiederhole eine String-Anweisung, solange das Zero-Flag gelöscht ist

Syntax: REPNE <String-Befehl> <Parameter>

Der REPNE-Befehl ist kein eigenständiger Maschinenbefehl, sondern ein sogenanntes »Wiederholungspräfix«. Sie setzen »REPNE« vor einen beliebigen String-Befehl, der je nach Syntax einige Parameter hat. Der Befehl hinter dem REPNE-Präfix wird dann so lange wiederholt, bis das CX-Register den Wert 0 besitzt. Wenn der String-Befehl eine der beiden Vergleichsoperationen CMPS oder SCAS ist, so bricht der Prozessor die Schleife auch dann ab, wenn das Zero-Flag gesetzt wird, er also ein identisches Byte, Wort oder Doppelwort bei den zu vergleichenden Strings findet. Nach jeder Ausführung des jeweiligen String-Befehls zählt der Prozessor das CX-Register um 1 herab.
Man benutzt den REPNE-Befehl in Schleifen, die ganze Zeichenketten verarbeiten sollen, zum Beispiel beim MOVS-Kommando. Bei diesen Befehlen tragen Sie zuerst in das CX-Register die Länge des Strings ein und verarbeiten dann mit dem REPNE-Präfix den kompletten String.
Beispiel:

```
MOV CX,100 ;der String ist 100 Worte lang
LEA SI,STRING1 ;lade Adresse des ersten Strings
LEA DI,ES:STRING2 ;lade Adresse des zweiten Strings
REPNE CMPSW ;vergleiche einzelne String-Worte in einer Schleife, bis der String beendet ist oder eine Übereinstimmung auftritt
```

REPNZ
»Repeat while not zero«, wiederhole eine String-Anweisung, solange das Zero-Flag gelöscht ist

Syntax: REPNZ <String-Befehl> <Parameter>

Der REPNZ-Befehl ist identisch mit dem REPNE-Kommando und stellt lediglich eine alternative Schreibweise dar.

SCAS
»Scan string«, durchsuche einen String nach einer bestimmten Zeichenfolge

Syntax: SCAS <Ziel-String>

Dieser Befehl dient zum Durchsuchen eines Strings nach einem bestimmten Inhalt. Der über ES:DI adressierte Ziel-String wird mit dem Inhalt des Akkumulators (AL, AX oder EAX) verglichen. Dazu subtrahiert der Prozessor das aktuelle String-Element vom Akkumulator und setzt die Kennzeichenbits abhängig vom Ergebnis der Differenz. Die Inhalte des Strings oder des Akkumulators werden dabei nicht verändert. Nach dem Vergleich wird das DI-Register je nach Datentyp und je nach Zustand des Richtungs-Flags um die Werte 1, 2 oder 4 erhöht oder vermindert.

Meist verwendet man den SCAS-Befehl mit einem Wiederholungspräfix, um eine ganze Zeichenkette nach einem bestimmten Wort, Byte oder Doppelwort zu durchsuchen.

Das SCAS-Kommando selbst stellt keinen gültigen Maschinenbefehl dar. Der Assembler verwendet den Ziel-String lediglich, um den Datentyp des Strings zu ermitteln, und übersetzt das SCAS-Kommando dann entweder in den SCASB-, den SCASW- oder den SCASD-Befehl.

Beispiel:

```
;ES muß bereits auf das Extrasegment zeigen
LEA DI,ES:STRING ;lade Offset eines Strings
MOV CX,500 ;String ist 500 Bytes lang
MOV AL,41h ;Vergleichselement ist der Buchstabe »A«
REPNZ SCAS STRING ;durchsuche den String
JCXZ NICHTDA ;CX ist jetzt 0, also wurde der Buchstabe nicht gefunden. Verzweige zum Label »NICHTDA«
... ;ansonsten ist der Buchstabe enthalten
NICHTDA: ... ;Buchstabe ist nicht enthalten
```

SCASB
»Scan string byte«, vergleiche ein Byte eines Strings mit dem Inhalt des AL-Registers

Syntax: SCASB

Dieses Kommando ist einer der drei Befehle, in die der SCAS-Befehl vom Assembler übersetzt wird. »SCASB« vergleicht ein Byte des über ES:DI adressierten Strings mit dem Inhalt des AL-Registers. Dazu subtrahiert der Prozessor den Inhalt des Strings an der Stelle ES:DI von AL und setzt die Kennzeichenbits entsprechend dem Ergebnis. ES und AL werden dabei nicht verändert. Nach dem Vergleich wird das DI-Register je nach Zustand des Richtungs-Flags um 1 erhöht oder vermindert.

Man verwendet den SCASB-Befehl meist in REP-Schleifen, um in einem String ein bestimmtes Byte zu suchen.

Beispiel:

```
;ES muß bereits auf das Extrasegment zeigen
LEA DI,ES:STRING ;lade Offset eines Strings
MOV CX,500 ;String ist 500 Bytes lang
MOV AL,42h ;Vergleichswert ist der Buchstaben »B«
REPNZ SCASB ;durchsuche String
JCXZ NICHTDA ;wenn jetzt CX = 0 ist, dann wurde der Buchstabe nicht gefunden – gehe zum Label »NICHTDA«
... ;Buchstabe ist enthalten
NICHTDA: ... ;Buchstabe ist nicht enthalten
```

SCASW

»Scan string word«, vergleiche ein Wort eines Strings mit dem Inhalt des AX-Registers

Syntax: SCASW

Dieses Kommando ist einer der drei Befehle, in die der SCAS-Befehl vom Assembler übersetzt wird. »SCASW« vergleicht ein Wort des mit ES:DI adressierten Strings mit dem Inhalt des AX-Registers. Dazu subtrahiert der Prozessor den Inhalt des Strings an der Position ES:DI von AX und setzt die Kennzeichenbits entsprechend dem Ergebnis. ES und AX werden dabei nicht verändert. Nach dem Vergleich wird das DI-Register je nach Zustand des Richtungs-Flags um 2 erhöht oder vermindert.
Man verwendet den SCASW-Befehl meist in REP-Schleifen, um in einem String ein bestimmtes Wort (zwei Byte) zu suchen.
Beispiel:

```
;ES muß bereits auf das Extrasegment
zeigen
LEA DI,ES:STRING ;lade Offset eines
Strings
MOV CX,500 ;String ist 500 Worte lang
MOV AX,0F23h ;Vergleichswert ist die Zahl
0F23hex
REPNZ SCASW ;durchsuche String
JCXZ NICHTDA ;wenn jetzt CX = 0 ist, dann
wurde die Zahl nicht gefunden. Gehe zum
Label »NICHTDA«
... ;Zahl ist enthalten
NICHTDA: ... ;Zahl ist nicht enthalten
```

SCASD

»Scan string double word«, vergleiche ein Doppelwort eines Strings mit dem Inhalt des EAX-Registers (nur 80386)

Syntax: SCASD

Dieses Kommando ist einer der drei Befehle, in die der SCAS-Befehl vom Assembler übersetzt wird; es ist nur beim 80386-Prozessor verfügbar. SCASD vergleicht ein Doppelwort des mit ES:DI adressierten Strings mit dem Inhalt des EAX-Registers. Dazu subtrahiert der Prozessor den Inhalt des Strings an der Position ES:DI von EAX und setzt die Kennzeichenbits entsprechend dem Ergebnis. ES und EAX werden dabei nicht verändert. Nach dem Vergleich wird das DI-Register je nach Zustand des Richtungs-Flags um 4 erhöht oder vermindert.
Man verwendet den SCASD-Befehl meist in REP-Schleifen, um in einem String ein bestimmtes Doppelwort (vier Byte) zu suchen.
Beispiel:

```
;ES muß bereits auf das Extrasegment
zeigen
LEA DI,ES:STRING ;lade Offset eines
Strings
MOV CX,500 ;String ist 500 Worte lang
MOV EAX,10F20F23h ;suche die Zahl
10F20F23hex
REPNZ SCASW ;durchsuche String
JCXZ NICHTDA ;wenn jetzt CX = 0 ist, dann
wurde die Zahl nicht gefunden - gehe zum
Label »NICHTDA«
... ;Zahl ist enthalten
NICHTDA: ... ;Zahl ist nicht enthalten
```

STD

»Set direction flag«, setze das Richtungskennzeichen

Syntax: STD

Über den Zustand des Richtungskennzeichens teilen Sie dem Prozessor mit, ob er sich bei mehrfacher Anwendung von String-Befehlen von den niedrigen Speicheradressen zu den höheren vorarbeitet, also den String »vorwärts« liest oder umgekehrt. Ein gesetztes Richtungskennzeichen bedeutet, daß die String-Zeiger SI und DI inkrementiert, ein gelöschtes Richtungskennzeichen, daß die String-Zeiger dekrementiert werden sollen.
Beispiel:

```
STD ;setze Richtungskennzeichen
```

STOS

»Store string«, kopiere einen String aus dem Akkumulator in den Speicher

Syntax: STOS <Ziel-String>

Mit diesem Befehl wird an der mit ES:DI adressierten Stelle des Ziel-Strings der Inhalt des Akkumulators eingetragen. Nach der Operation wird DI je nach String-Typ und je nach Zustand des Richtungs-Flags um die Werte 1, 2 oder 4 erhöht oder vermindert.
Das Kommando eignet sich vor allem im Zusammenhang mit einem Wiederholungspräfix dazu, bestimmte Teile eines Strings oder einen kompletten String mit einem bestimmten Inhalt zu füllen.
»STOS« selbst stellt allerdings keinen gültigen Maschinenbefehl dar. Der Parameter Ziel-String wird vom Assembler nur dazu benutzt, um den Datentyp des Strings festzustellen und den STOS-Befehl dann entweder in das STOSB-, STOSW- oder STOSD-Kommando zu übersetzen.
Beispiel:

```
;ES muß bereits auf das Extrasegment
zeigen
LEA DI,ES:STRING ;lade Offset eines
Strings
MOV CX,50 ;50 Worte sollen gefüllt werden
MOV AX,1000h ;und zwar mit 1000hex
REPE STOS STRING ;führe Operation aus
```

STOSB

»Store string byte«, kopiere ein Byte aus dem AL-Register in einen String

Syntax: STOSB

Dieser Befehl ist einer der drei Kommandos, in die der Assembler den STOS-Befehl übersetzt. Mit dem STOSB-Befehl wird der Inhalt des AL-Registers an die mit ES:DI festgelegte Position eines Strings übertragen.
Nach der Operation wird der Zeiger DI je nach Zustand des Richtungskennzeichens um 1 erhöht oder vermindert.
Beispiel:

```
;ES muß bereits auf das Extrasegment
zeigen
LEA DI,ES:STRING ;lade Offset eines
Strings
MOV CX,50 ;50 Bytes sollen gefüllt werden
MOV AL,'A' ;und zwar mit dem Buchstaben
»A«
REPE STOSB ;führe Operation aus
```

STOSW

»Store string word«, kopiere ein Wort aus dem AX-Register in einen String

Syntax: STOSW

Dieser Befehl ist eines der drei Kommandos, in die der Assembler den STOS-Befehl übersetzt. Mit dem STOSW-Befehl wird der Inhalt des AX-Registers an die mit ES:DI festgelegte Position eines Strings übertragen. Nach der Operation wird der Zeiger DI je nach Zustand des Richtungskennzeichens um 2 erhöht oder vermindert.
Beispiel:

```
;ES muß bereits auf das Extrasegment
zeigen
LEA DI,ES:STRING ;lade Offset eines
Strings
MOV CX,50 ;50 Worte sollen gefüllt werden
MOV AX,0 ;und zwar mit Null
REPE STOSW ;führe Operation aus
```

STOSD

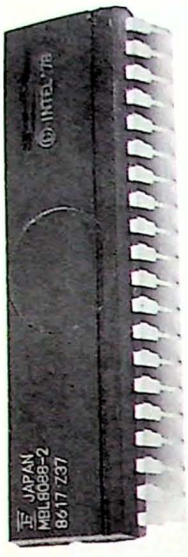
»Store string double word«, kopiere ein Doppelwort aus dem EAX-Register in einen String

Syntax: STOSD

Dieser Befehl ist eines der drei Kommandos, in die der Assembler den STOS-Befehl übersetzt; er ist nur beim 80386-Prozessor verfügbar. Mit dem STOSD-Befehl wird der Inhalt des EAX-Registers an die mit ES:DI festgelegte Stelle eines Strings übertragen.
Nach der Operation wird der Zeiger DI je nach Zustand des Richtungskennzeichens um 4 erhöht oder vermindert.
Beispiel:

```
;ES ist schon auf das Extrasegment ini-
tialisiert
LEA DI,ES:STRING ;lade Offset eines
Strings
MOV CX,50 ;50 Worte sollen gefüllt werden
MOV EAX,0 ;und zwar mit Null
REPE STOSD ;führe Operation aus
```

(Martin Althaus/tr)



Assemblerreferenz:
Bitmanipulation und Logik

Diese Übersicht beschreibt alle Kommandos, die logische Operationen durchführen oder gezielt bestimmte Kennzeichen (Flags) des Prozessors verändern. Als Einsatzgebiete kommen häufig Aufgaben in Betracht, bei denen bestimmte Bits eines Wertes getestet oder modifiziert werden müssen.

Man kann die Kommandos der Gruppe »Bitmanipulationen und Logik« in zwei Bereiche aufteilen: Auf der einen Seite gibt es die Befehle zur bitweisen, logischen Verknüpfung zweier Werte und auf der anderen Seite die Kommandos zum bitweisen Verschieben oder Rotieren eines Wertes. Hinzu kommen einige Sonderbefehle für das Testen von Bits und das bedingte Setzen von Bytes, die nur beim 80386-Prozessor verfügbar sind.

AND
»Logical AND«, logische Und-Verknüpfung

Syntax: AND <Zieloperand>,<Quelloperand>

Der AND-Befehl führt eine bitweise, logische Und-Verknüpfung zwischen dem Quelloperanden und dem Zieloperanden durch. Dazu werden die Bits der beiden Operanden der Reihe nach miteinander verknüpft. Sind zwei korrespondierende Bits in beiden Operanden gesetzt, dann setzt der Befehl auch das entsprechende Bit im Ergebnis. Die Verknüpfungstabelle für zwei Bits sieht so aus:

```
0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1
```

Als Operanden können Sie zwei Register, ein Register und eine Speicherstelle oder eine Konstante und ein Register oder eine Speicherstelle verknüpfen. Der 80386-Prozessor erlaubt auch Operanden mit einer Länge von 32 Bit.
Dieses Kommando wird vor allem dazu eingesetzt, bestimmte Bits eines Operanden zu löschen (»auszublenden«). Sie löschen ein beliebiges Bit, indem Sie den Operanden mit einem Wert verknüpfen, bei dem alle Bits bis auf das zu löschende gesetzt sind.
Beispiel:

```
MOV DX,03BFh ;übertrage eine Port-Adresse nach DX
IN AL,DX ;lese ein Byte aus diesem Port
AND AL,01111111b ;blende das achte Bit aus
OUT AL,DX ;und schreibe das Ergebnis zurück zum Port
```

BSF
»Bit scan forward«, Bitsuchlauf vorwärts (von »rechts« nach »links«)

Syntax: BSF <Zieloperand>,<Quelloperand>

Der BSF-Befehl ist nur auf dem 80386-Prozessor verfügbar. Der Rechner untersucht das mit dem Quelloperanden angegebene Bitmuster von rechts nach links so lange, bis er ein gesetztes Bit findet. Im Zieloperanden wird dann dieses eine Bit gesetzt und alle anderen gelöscht.
Als Operanden sind zwei Register oder ein Register und ein Speicherinhalt zulässig. Beide Operanden müssen allerdings den gleichen Datentyp (16 oder 32 Bit) besitzen.
Beispiel:

```
MOV BX,SPEICHER1 ;lade Speicherinhalt nach BX
BSF SPEICHER2,BX ;suche gesetztes Bit und speichere es in einer anderen Adresse
```

BSR
»Bit scan reverse«, Bitsuchlauf rückwärts (von »links« nach »rechts«)

Syntax: BSR <Zieloperand>,<Quelloperand>

Dieser Befehl ist nur beim 80386-Prozessor verfügbar und funktioniert analog dem BSF-Kommando, arbeitet aber von links nach rechts. Das mit dem Quelloperanden angegebene Bitmuster wird von links nach rechts bis zum ersten gesetzten Bit durchsucht. Im Zieloperanden wird dann dieses eine Bit gesetzt und alle anderen gelöscht.
Als Operanden sind zwei Register oder ein Register und ein Speicherinhalt zulässig. Beide Operanden müssen allerdings im gleichen Datentyp (16 oder 32 Bit) vorliegen.
Beispiel:

```
MOV BX,SPEICHER1 ;lade Speicherinhalt nach BX
BSR SPEICHER2,BX ;suche gesetztes Bit und lege es an einer anderen Adresse ab
```

BT
»Bit test«, untersuche Bits

Syntax: BT <Quelloperand>,<Maskenoperand>

Dieser Befehl ist nur beim 80386-Prozessor verfügbar. Mit dem BT-Kommando werden alle im Maskenoperanden gesetzten Bits mit denjenigen im Quelloperanden verglichen. Ist mindestens ein Bit in beiden Werten gesetzt, dann setzt der Prozessor das Übertragskennzeichen (»Carry-Flag«) auf Eins.
Als Operanden sind 16- oder 32-Bit-Werte zulässig. Es dürfen zwei Register miteinander verglichen werden oder ein Register und eine Speicheradresse oder ein Register oder eine Speicheradresse und eine Konstante oder eine Speicheradresse und ein Register oder eine Konstante.
Beispiel:

```
MOV BL,10101010b ;lade Binärzahl nach BL
MOV AL,00000011b ;lade Maske nach AL
BT BL,AL ;das Carry-Flag ist jetzt gesetzt (Bit 1 ist in beiden Werten gesetzt), und der Wert in BL bleibt unverändert
```

BTC
»Bit test and complement«, untersuche Bits und bilde Komplementärform

Syntax: BTC <Quelloperand>,<Maskenoperand>

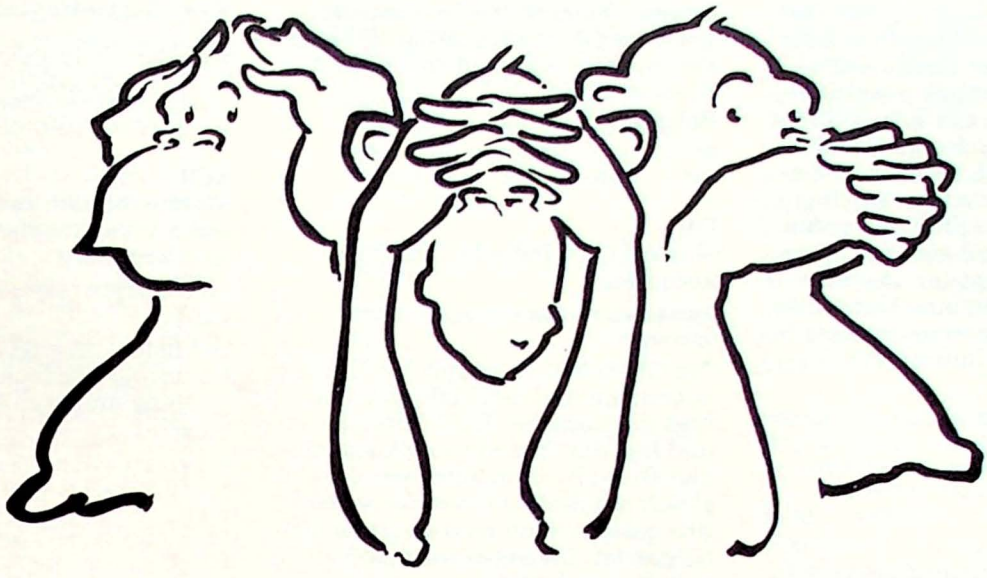
Dieser Befehl ist nur beim 80386-Prozessor verfügbar. Mit dem BTC-Kommando werden alle im Maskenoperanden gesetzten Bits mit denjenigen im Quelloperanden verglichen. Ist mindestens ein Bit in beiden Werten gesetzt, dann setzt der Prozessor das Übertragskennzeichen (»Carry-Flag«) auf Eins. Zusätzlich werden im Quelloperanden alle Bits invertiert, die im Maskenoperanden gesetzt sind.
Als Operanden sind 16- oder 32-Bit-Werte zulässig. Es dürfen zwei Register miteinander verglichen werden, oder ein Register und eine Speicheradresse oder ein Register oder eine Speicheradresse und eine Konstante oder eine Speicheradresse und ein Register oder eine Konstante.
Beispiel:

```
MOV BL,10101010b ;lade Binärzahl nach BL
MOV AL,00000011b ;lade Maske nach AL
BT BL,AL ;das Carry-Flag ist jetzt gesetzt (Bit 1 ist in beiden Werten gesetzt), und in BL steht 10101001b
```

BTR
»Bit test and reset«, überprüfe Bits und setze sie auf Null

Syntax: BTR <Quelloperand>,<Maskenoperand>

Dieser Befehl ist nur beim 80386-Prozessor verfügbar. Mit dem BTR-Kommando werden alle im Maskenoperanden gesetzten Bits mit denjenigen im Quelloperanden verglichen.



KLIMAVERÄNDERUNG, WIESO ? HAUPTSACHE DIE SONNE SCHEINT !

Irrtum. Es könnte Ihnen bald zu heiß werden.

Die meisten Menschen wollen nur schönes Wetter. Und merken nichts von der allmählichen Veränderung des Weltklimas. Obwohl jeder davon betroffen sein wird. Schon jetzt gibt es genügend Gründe, sich mit den Folgen auseinanderzusetzen.

Wenn Sie wissen wollen, warum das Wetter immer merkwürdiger wird und welche Konsequenzen das für Sie hat, dann füllen Sie den Coupon aus. Wir beantworten Ihre Fragen, sagen Ihnen, wo wir uns noch engagieren, warum und wie Sie uns dabei helfen können.

Ja, ich will von ROBIN WOOD wissen, warum das Wetter bei uns immer merkwürdiger wird und was das für Folgen hat.

Name: _____

Straße: _____

Wohnort: _____

dmv

Weil ich helfen will, lege ich 3 Mark in Briefmarken bei. Coupon einsenden an ROBIN WOOD, Postfach 10 21 22, 2800 Bremen 1

ROBIN WOOD

Gewaltfreie Aktionsgemeinschaft für Natur und Umwelt e.V.

Ist mindestens ein Bit in beiden Werten gesetzt, dann setzt der Prozessor das Übertragskennzeichen (»Carry-Flag«) auf Eins. Zusätzlich werden im Quelloperanden alle Bits gelöscht, die im Maskenoperanden gesetzt sind. Als Operanden sind 16- oder 32-Bit-Werte zulässig. Es dürfen zwei Register miteinander verglichen werden oder ein Register und eine Speicheradresse oder ein Register oder eine Speicheradresse und eine Konstante oder eine Speicheradresse und ein Register oder eine Konstante.
Beispiel:

```
MOV BL,10101010b ;lade Binärzahl nach BL
MOV AL,00000011b ;lade Maske nach AL
BT BL,AL ;das Carry-Flag ist jetzt gesetzt (Bit 1 ist in beiden Werten gesetzt), und in BL steht 10101000b
```

BTS
»Bit test and set«, überprüfe Bits und setze sie auf Eins

```
Syntax: BTS <Quelloperand>,<Maskenoperand>
```

Dieser Befehl ist nur beim 80386-Prozessor verfügbar. Mit dem BTS-Kommando werden alle im Maskenoperanden gesetzten Bits mit denjenigen im Quelloperanden verglichen. Ist mindestens ein Bit in beiden Werten gesetzt, dann setzt der Prozessor das Übertragskennzeichen (»Carry-Flag«) auf Eins. Zusätzlich werden im Quelloperanden alle Bits gesetzt, die im Maskenoperanden gesetzt sind.
Als Operanden sind 16- oder 32-Bit-Werte zulässig. Es dürfen zwei Register miteinander verglichen werden oder ein Register und eine Speicheradresse oder ein Register oder eine Speicheradresse und eine Konstante oder eine Speicheradresse und ein Register oder eine Konstante.
Beispiel:

```
MOV BL,10101010b ;lade Binärzahl nach BL
MOV AL,00000011b ;lade Maske nach AL
BT BL,AL ;das Carry-Flag ist jetzt gesetzt (Bit 1 ist in beiden Werten gesetzt), und in BL steht 10101011b
```

NOT
»Logical NOT«, logische Nicht-Verknüpfung

```
Syntax: NOT <Zieloperand>
```

Dieser Befehl bildet das logische Gegenteil eines Operanden. Das NOT-Kommando dreht jedes Bit des Zieloperanden um und speichert das Ergebnis wieder im Zieloperanden. Die Logiktabelle für eine NOT-Verknüpfung sieht folgendermaßen aus:

```
NOT 0 = 1
NOT 1 = 0
```

Jedes vor dem Aufruf von »NOT« gesetzte Bit im Zieloperanden ist also nach dem Aufruf gelöscht, jedes vorher gelöschte Bit gesetzt. Das NOT-Kommando erwartet als Zieloperanden ein Register oder eine Speicheradresse. Beim 80386-Prozessor dürfen diese Operanden bis zu 32 Bit besitzen, beim 8086 und 80286 nur 8 Bit oder 16 Bit.
Beispiel:

```
MOV AL,10101010b ;lade Wert nach AL
NOT AL ;bilde daraus 01010101b
```

OR
»logical OR«, logische Oder-Verknüpfung

```
Syntax: OR <Zieloperand>,<Quelloperand>
```

Der OR-Befehl verknüpft den Quelloperanden und den Zieloperanden über die logische ODER-Funktion und legt das Ergebnis im Zieloperanden ab. Ist bei dem bitweisen Vergleich mindestens eines der beiden Bits gesetzt, dann wird auch das Zielbit gesetzt. Die entsprechende Verknüpfungstabelle sieht so aus:

```
0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1
```

Als Operanden können zwei Register miteinander, ein Register und eine Speicherstelle oder eine Konstante und ein Register oder eine Speicherstelle verknüpft werden. Der 80386--Prozessor erlaubt auch Operanden mit 32 Bit Länge.
Der OR-Befehl wird häufig eingesetzt, um in einem Operanden ein oder mehrere Bits zu setzen und die restlichen Bits unverändert zu übernehmen. Dazu setzen Sie im Quelloperanden die gewünschten Bits und führen eine logische Oder-Funktion aus.
Beispiel:

```
MOV DX,03BFh ;lade eine Port-Adresse nach DX
IN AL,DX ;lese ein Byte aus dem Port
OR AL,10000000b ;setze das achte Bit
OUT AL,DX ;und schreibe Wert zurück zum Port
```

RCL
»Rotate through carry left«, rotiere einen Wert bitweise nach links über das Carry-Flag

```
Syntax: RCL <Zieloperand>,<Bitanzahl>
```

Der RCL-Befehl verschiebt alle Bits im Zieloperanden um eine Position nach links. Der Inhalt des Carry-Flags wird jeweils in das Bit 0 übertragen und das höchste Bit des Zieloperanden ins Carry-Flag kopiert. Der Prozessor führt den Vorgang so oft aus, wie durch »Bitanzahl« angegeben ist (dies kann auch der Inhalt des CL-Registers sein).
Der Zieloperand darf ein Register oder eine Speicheradresse sein, beim 80386-Prozessor auch im 32-Bit-Format.
Beispiel:

```
MOV AL,11110000b ;lade einen Wert nach AL
MOV CL,3 ;bereite dreimaliges Schieben vor
STC ;setze Carry-Flag
RCL AL,CL ;schiebe AL »CL-mal« nach links. Das Carry-Flag ist danach gesetzt und in AL steht 10000111b
```

RCR
»Rotate through carry right«, rotiere einen Wert bitweise nach rechts über das Carry-Flag

```
Syntax: RCR <Zieloperand>,<Bitanzahl>
```

Der Befehl arbeitet analog zum RCL-Kommando, jedoch ist die Schieberichtung umgekehrt.
Beispiel:

```
MOV AL,11110000b ;lade einen Wert nach AL
MOV CL,3 ;bereite dreimaliges Schieben vor
STC ;setze Carry-Flag
RCR AL,CL ;schiebe AL »CL-mal« nach rechts. Das Carry-Flag ist danach gelöscht und in AL steht 00111110b
```

ROL
»Rotate left«, rotiere einen Wert bitweise nach links

```
Syntax: ROL <Zieloperand>,<Bitanzahl>
```

Der Inhalt des Zieloperanden wird jeweils bitweise nach links verschoben. Bei jedem Vorgang kopiert der Prozessor das höchstwertige Bit in das Bit 0. Die Anzahl der Verschiebungen legen Sie durch den Operanden »Bitanzahl« fest, der entweder eine Konstante oder das CL-Register angibt.
Der Zieloperand darf ein Register oder eine Speicheradresse sein, beim 80386-Prozerssor auch im 32-Bit-Format.
Beispiel:

```
MOV AL,11110000b ;lade einen Wert nach AL
MOV CL,3 ;bereite dreimaliges Schieben vor
STC ;setze Carry-Flag
ROL AL,CL ;schiebe AL »CL-mal« nach links. In AL steht danach 10000111b
```

ROR
»Rotate Right«, verschiebe einen Wert bitweise nach rechts

```
Syntax: ROR <Zieloperand>,<Bitanzahl>
```

Der ROR-Befehl arbeitet analog zu »ROL«, lediglich die Schieberichtung ist umgekehrt.
Beispiel:

```
MOV AL,11110000b ;lade einen Wert nach AL
MOV CL,3 ;bereite dreimaliges Schieben vor
STC ;setze Carry-Flag
ROR AL,CL ;schiebe AL »CL-mal« nach rechts. In AL steht danach 00011110b
```

SAL
»Shift arithmetic left«, arithmetische, bitweise Verschiebung nach links

```
Syntax: SAL <Zieloperand>,<Bitanzahl>
```

Der SAL-Befehl verschiebt den Inhalt des Zieloperanden um die angegebene Anzahl an Bits nach links, wobei von rechts Nullen nachgeschoben werden und die links herausfallenden Bits verlorengehen.
Der Zieloperand darf ein Register oder eine Speicheradresse sein, beim 80386-Prozessor auch im 32-Bit-Format. Die »Bitanzahl« ist entweder eine Konstante oder das CL-Register. Sie können diesen Befehl sehr gut dazu verwenden, eine schnelle Multiplikationen mit Zweierpotenzen durchzuführen. Jede bitweise Verschiebung eines Wertes nach links entspricht einer Multiplikation mit dem Wert 2.
Beispiel:

```
MOV AL,00001111b ;lade Wert 15 nach AL
MOV CL,3 ;lade 3 nach CL
SAL AL,CL ;schiebe AL dreimal nach links.
Das Ergebnis ist 120 (01111000b) und entspricht einer Multiplikation mit der Zahl
2^3 (8)
```

SAR
»Shift arithmetic right«, arithmetische, bitweise Verschiebung nach rechts

Syntax: SAR <Zieloperand>,<Bitanzahl>

Der SAR-Befehl entspricht dem SAL-Kommando, nur ist die Schieberichtung umgekehrt. Von links werden Nullen nachgeschoben, die rechts herausfallenden Bits gehen verloren. Jede einzelne Verschiebung entspricht einer Division durch die Zahl 2, wobei ein eventueller Divisionsrest wegfällt.
Beispiel:

```
MOV AL,00010000b ;lade Wert 16 nach AL
MOV CL,3 ;lade 3 nach CL
SAL AL,CL ;schiebe AL dreimal nach
rechts. Das Ergebnis ist 2 (00000010b)
und entspricht einer Division durch die
Zahl 2^3 (8)
```

SETcc
»Set byte on condition code«, setze ein Byte abhängig von einer bestimmten Bedingung auf Eins

Syntax: SETcc <Zieloperand>

Dieses Kommando ist nur auf dem 80386-Prozessor verfügbar. Es setzt abhängig von der mit »cc« formulierten Bedingung den Zieloperanden auf Eins (logisch »wahr«), falls die Bedingung auf das Ergebnis der letzten Vergleichsoperation zutrifft. Der 80386-Prozessor kennt die folgenden Bedingungen, die anstelle des Parameters »cc« einzusetzen sind:

B	kleiner als
NAE	nicht größer und nicht gleich
C	Carry-Flag gesetzt
BE	kleiner als oder gleich
NA	nicht größer
E	gleich Null
Z	gleich Null
L	kleiner als
NGE	nicht größer und nicht gleich
LE	kleiner oder gleich
NG	nicht größer
NB	nicht kleiner
AE	größer oder gleich
NC	Carry-Flag nicht gesetzt
NBE	nicht kleiner und nicht gleich
A	größer
NE	ungleich Null
NZ	ungleich Null
NL	nicht kleiner
GE	größer oder gleich
NLE	nicht kleiner und nicht gleich
G	größer
NO	kein Überlauf (gelöschtes Overflow-Flag)
NP	keine Parität
PO	Parität ungerade
NS	kein Vorzeichen
O	Überlauf (gesetztes Overflow-Flag)
P	Parität
PE	Parität gerade
S	Vorzeichen

Um auf die entsprechende Bedingung zu prüfen, setzen Sie die Buchstabenkombination im Befehl hinter den Buchstaben »SET« ein (also zum Beispiel »SETNE«). Der Zieloperand darf ein 8-Bit-Register oder eine Speicheradresse (Byte) sein.
Beispiel:

```
MOV AX,0 ;setze AX auf Null
SETZ SPEICHER ;Der Inhalt der Adresse
»SPEICHER« wird auf Eins gesetzt, da die
letzte Operation Null ergab
```

SHL
»Shift left«, verschiebe einen Wert bitweise nach links

Syntax: SHL <Zieloperand>,<Bitanzahl>

Dieser Befehl ist identisch mit dem SAL-Kommando und stellt nur eine alternative Schreibweise dar. Der Assembler übersetzt SAL und SHL in den gleichen Maschinencode.

SHR
»Shift right«, verschiebe einen Wert bitweise nach rechts

Syntax: SHR <Zieloperand>,<Bitanzahl>

Dieser Befehl ist identisch mit dem SAR-Kommando und stellt nur eine alternative Schreibweise dar. Der Assembler übersetzt SAR und SHR in den gleichen Maschinencode.

TEST
»logical compare«, bitweiser Vergleich zweier Werte

Syntax: TEST <Quelloperand>,<Maskenoperand>

Dieser Befehl veranlaßt eine logische Und-Verknüpfung zwischen dem Quelloperanden und dem Maskenoperanden. Das Ergebnis des Vergleichs wird jedoch nicht gespeichert, es dient lediglich dazu, die Kennzeichenbits (Flags) des Prozessors zu aktualisieren.
Als Operanden dürfen Sie zwei Register, ein Register und eine Speicheradresse oder ein Register und eine Konstante miteinander vergleichen.
Man benutzt diesen Befehl vor allem, um zu überprüfen, ob bestimmte Bits eines Operanden gesetzt sind. In einem solchen Fall setzen Sie als Quelloperand den zu testenden Wert und als Maskenoperand ein Bitmuster ein, bei dem alle Bits bis auf die zu testenden gelöscht sind. Läßt man auf den TEST-Befehl eine JNZ-Verzweigung folgen, dann wird diese immer dann ausgeführt, wenn es mindestens eine Übereinstimmung bei den Bits gegeben hat.
Beispiel:

```
MOV DX,03BFh ;lade Portadresse 3BFhex
IN AL,DX ;lese Wert von Port
TEST AL,128 ;Bit 8 gesetzt?
JNZ BIT7 ;ja: springe zum Label BIT8
```

XOR
»Exclusive or«, logische Exklusiv-Oder-Verknüpfung

Syntax: XOR <Zieloperand>,<Quelloperand>

Dieser Befehl bildet die logische, ausschließende Oder-Verknüpfung aus zwei gegebenen Operanden. Dazu werden die Bits der beiden Operanden der Reihe nach miteinander verknüpft. Besitzen je zwei korrespondierende Bits in beiden Operanden einen unterschiedlichen Wert, wird das entsprechende Bit auch im Ergebnis gesetzt, ansonsten (also wenn beide Bits gleich sind) wird das entsprechende Bit im Ergebnis gelöscht. Die Verknüpfungstabelle sieht folgendermaßen aus:

```
0 XOR 0 = 0
0 XOR 1 = 1
1 XOR 0 = 1
1 XOR 1 = 0
```

Als Operanden können Sie zwei Register einsetzen oder ein Register und eine Speicheradresse oder eine Konstante oder ein Register und eine Speicherstelle. Der 80386-Prozessor erlaubt auch Operanden mit 32 Bit Länge.
Man verwendet dieses Kommando häufig, um einem Register den Wert 0 zuzuweisen. Dazu wird das entsprechende Register über den XOR-Befehl mit sich selbst verknüpft. Weil dann alle Bits den gleichen Inhalt haben, ist das Ergebnis Null. Diese Methode arbeitet wesentlich schneller als das Laden einer Null in das Register mit Hilfe des MOV-Kommandos, da hierzu keine Speicherzugriffe notwendig sind.
Der XOR-Befehl läßt sich auch dazu einsetzen, um einen Wert zu invertieren. Dazu verknüpfen Sie diesen mit einer Zahl, bei der alle Bits gesetzt sind. Allerdings bietet das NOT-Kommando hierzu eine wesentlich schnellere Lösung.
Beispiel:

```
XOR AX,AX ;lösche AX-Register
```

(Martin Althaus/tr)

Assemblerreferenz: Arithmetik

Im Gegensatz zu manch anderen Prozessoren (zum Beispiel 6502 oder Z80) verfügen die Intel-Prozessoren über eine Vielzahl leistungsfähiger Befehle für die schnelle und vor allem einfache Berechnung arithmetischer Formeln. Die dafür zuständigen Kommandos finden Sie in der folgenden Übersicht.

Neben den »normalen« Befehlen rund um die arithmetischen Operationen »Addition«, »Subtraktion«, »Multiplikation« und »Division« beschäftigt sich ein großer Teil der Arithmetik-Befehle mit der Umwandlung von Binärzahlen ins BCD-Format und umgekehrt.

Bei den sogenannten »binär codierten Dezimalzahlen« (aus dem Englischen: »binary coded decimal«, kurz »BCD«) wird eine zweistellige Dezimalzahl in ihre beiden Ziffern (Zehner und Einer) aufgeteilt. Jede dieser dezimalen Ziffern kann einen Wert zwischen null und neun annehmen. Der Wert wird bei BCD-Zahlen in je einem halben Byte untergebracht, wobei die untere Hälfte des Bytes die niederwertige Ziffer und die obere Hälfte die höherwertige Ziffer enthält. Dieses Format bezeichnet man auch als »gepackte BCD-Kodierung« im Gegensatz zum ungepackten Format, bei dem je eine Ziffer in einem Byte untergebracht wird, also jeweils ein halbes Byte unbenutzt bleibt.

In der hexadezimalen Schreibweise von Zahlen läßt sich die BCD-Codierung gut verdeutlichen, weil jede Hexadezimalziffer genau ein halbes Byte belegt. Möchte man zum Beispiel die Dezimalzahl 67 als gepackte BCD-Zahl formulieren, so lautet der dazu erforderliche Wert

$6 \times 16^1 + 7 \times 16^0 =$
$6 \times 16 + 7 =$
67hex

Nun ist es aber für den Prozessor bei arithmetischen Operationen nicht erkennbar, ob es sich um BCD- oder Binärzahlen handelt, die er verarbeiten soll. Daher treten bei der Verwendung von BCD-Zahlen mit den für Binärzahlen ausgerichteten Operationen Fehler auf. Will man beispielsweise die BCD-Zahl 39 von der BCD-Zahl 70 abziehen, sollte das Ergebnis die BCD-Zahl 31 sein. Verwendet man nun aber die gewohnte Assemblersequenz

```
MOV BL,39h ;BCD-Zahl 39 nach BL
MOV AL,70h ;BCD-Zahl 70 nach AL
SUB AL,BL ;beide Zahlen voneinander abziehen
```

dann ergibt sich ein falsches Ergebnis: Weil der Prozessor nämlich binär subtrahiert, bildet er die Differenz 112 (70hex) minus 57 (39hex). Das Resultat 55 (37hex) ist die BCD-Zahl 37.

Solche Rechenfehler bei Operationen mit BCD-Zahlen werden von einer Gruppe von Befehlen korrigiert, den sogenannten Adjust-Befehlen. Für jede der arithmetischen Prozessor-Operationen (Addition, Subtraktion, Multiplikation und Division) gibt es einen eigenen Adjust-Befehl, der das Ergebnis nach oder vor der Operation korrigiert, falls mit BCD-Zahlen gearbeitet wird.

AAA

»ASCII adjust after addition«, korrigiere das Ergebnis der Addition zweier ungepackter BCD-Zahlen

Syntax: AAA

Dieser Befehl dient zur Anpassung des Ergebnisses einer Addition von zwei BCD-Zahlen. Da der Prozessor BCD-Zahlen wie gewöhnliche Binärzahlen addiert, treten Fehler in den Ergebnissen auf. Das AAA-Kommando wandelt das Ergebnis der Addition zweier ungepackter BCD-Zahlen wieder in eine ungepackte BCD-Zahl um. Dabei wird davon ausgegangen, daß sich das Additionsergebnis im AX-Register (also eine Ziffer in AH und eine in AL) befindet. Damit der Befehl richtig arbeitet, sollte man ihn unmittelbar auf eine BCD-Addition folgen lassen, weil der Zustand der Prozessor-Flags bei der Ausführung des Befehls von Bedeutung ist.

Beispiel:

```
ADD AL,CH ;addiere ungepackte BCD-Zahlen in AL und CH
AAA ;justiere das Ergebnis auf BCD-Code
```

AAD

»ASCII adjust before division«, bereite eine ungepackte BCD-Zahl für eine Division vor

Syntax: AAD

Dieses Kommando dient zur Benutzung der Divisionsbefehle im Zusammenhang mit BCD-Zahlen. Da der Prozessor BCD-Zahlen wie gewöhnliche Binärzahlen dividiert, treten Fehler in den Ergebnissen auf. Der AAD-Befehl wandelt eine ungepackte BCD-Zahl im AX-Register (also eine Ziffer in AH und eine in AL) in einen Binärwert in AL um. Zur Division verwenden Sie dazu zweckmäßigerweise den IDIV-Befehl (vorher müssen Sie dafür sorgen, daß AH den Wert Null erhält, sonst ergibt die Division kein korrektes Ergebnis).

Beispiel:

```
MOV AX,0405h ;lade AX mit der ungepackten BCD-Zahl 45
MOV BL,5 ;lade BL mit dem Divisor 5
AAD ;wandle AX von BCD nach binär
IDIV BL ;dividiere AL durch BL
```

AAM

»ASCII adjust after multiply«, korrigiere das Ergebnis der Multiplikation zweier ungepackter BCD-Zahlen

Syntax: AAM

Dieser Befehl wandelt das Ergebnis der Multiplikation zweier BCD-Zahlen wieder in das BCD-Format zurück. Weil der Prozessor BCD-Zahlen wie gewöhnliche Binärzahlen multipliziert, treten bei den Ergebnissen Fehler auf. Mit dem AAM-Kommando werden diese Fehler ausgeglichen. Der im AL-Register enthaltene Wert wird bei Aufruf des Befehls in zwei ungepackte BCD-Ziffern unterteilt: Die höherwertige Ziffer kommt in das AH-Register, die niederwertige Ziffer in das AL-Register.

Beispiel:

```
MOV AX,0043h ;Lade 67 in AL und lösche AH
AAM ;wandle in BCD-Zahl um, nun steht in AH 06hex und in AL 07hex
```

AAS

»ASCII adjust after subtraction«, korrigiere das Ergebnis der Subtraktion zweier ungepackter BCD-Zahlen

Syntax: AAS

Mit diesem Kommando wird das Ergebnis einer Subtraktion von zwei BCD-Zahlen korrigiert. Da der Prozessor BCD-Zahlen wie gewöhnliche Binärzahlen subtrahiert, treten in den Ergebnissen arithmetische Fehler auf. Diese Fehler werden mit dem AAS-Befehl korrigiert. Der Befehl kann lediglich mit 8-Bit-Werten arbeiten, er verändert den Wert von AL nach einer erfolgten Subtraktion in eine gültige BCD-Zahl.

Beispiel:

```
MOV AL,53h ;lade BCD 53 nach AL
MOV CL,49h ;lade BCD 49 nach CL
SUB AL,CL ;bilde AL minus CL (10hex)
AAS ;korrigiere Ergebnis
```

ADC
»Add with carry«, addiere unter Berücksichtigung der Übertragsmarke »Carry-Flag«

Syntax: ADC <Zieloperand>,<Quelloperand>

»ADC« addiert zwei Zahlen (Worte oder Bytes). Der Zieloperand wird zum Quelloperanden addiert und das Ergebnis im Zieloperanden abgelegt. Es können Speicherinhalte zu Registern addiert werden und umgekehrt oder unmittelbare Werte zu Registern oder Speicherinhalten.
Bei der Addition wird ein eventueller Übertrag im Carry-Flag berücksichtigt. In der Praxis heißt das, es wird zusätzlich eine Eins addiert, wenn das Carry-Flag gesetzt ist, ansonsten nicht. Man verwendet diesen Befehl immer dann, wenn vor der Addition eine Operation ausgeführt wurde, die eventuell das Ergebnis der Addition beeinflussen könnte. Sollen zum Beispiel zwei 32-Bit-Operanden addiert werden, dann können Sie die Addition in zwei Schritten ausführen. Zuerst werden die unteren beiden Worte addiert (ohne Übertrag, also mit dem ADD-Kommando oder mit »ADC« und gelöschtem Carry-Flag). Dann addieren Sie die oberen beiden Worte mit dem ADC-Befehl. Dabei geht ein eventueller Übertrag aus der ersten Summe mit in die zweite Summe ein.
Beispiel:

```
;32-Bit-Addition
MOV AX,LOW_WORT1 ;lade unteres Wort1
MOV BX,LOW_WORT2 ;lade unteres Wort2
ADD AX,BX ;addiere ohne Carry
MOV CX,HIGH_WORT1 ;lade oberes Wort1
MOV DX,HIGH_WORT2 ;lade oberes Wort2
ADC CX,DX ;addiere, berücksichtige möglichen Übertrag
```

ADD
»Add«, addiere

Syntax: ADD <Quelloperand>,<Zieloperand>

Der Zieloperand wird zum Quelloperanden addiert und das Ergebnis im Zieloperanden abgelegt. Es können Speicherinhalte zu Registern addiert werden und umgekehrt oder unmittelbare Werte zu Registern oder Speicherinhalten.
Dieses Kommando addiert zwei Worte oder zwei Byte, ohne dabei (im Gegensatz zu »ADC«) einen Übertrag zu berücksichtigen. Es hat die gleiche Auswirkung wie der ADC-Befehl mit einem vorausgehenden Löschen des Übertrag-Flags (bis auf die Tatsache, daß das Löschen des Carry-Flags zwei zusätzliche Taktzyklen benötigt).
Beispiel:

```
MOV CX,100h ;lade CX mit 100hex
ADD CX,TABLE[SI] ;addiere einen Wert aus einer Tabelle
```

CBW
»Convert byte to word«, wandle ein Byte in ein Wort um

Syntax: CBW

Der CBW-Befehl wandelt das Byte im AL-Register in ein komplettes Wort im AX-Register um. Der Befehl ist nur bei vorzeichenbehafteten Binärzahlen sinnvoll, da hier das Vorzeichen vom Byte AL auch im Wort AX erhalten bleibt. Wenn AL also kleiner ist als 80hex, dann erhält AH den Wert 00, sonst FFhex.
Beispiel:

```
MOV AL,100 ;lade AL mit 100
NEG AL ;bilde -100
CBW ;und verwandle in Wort (ergibt FF64hex)
```

CLC
»Clear carry«, lösche die Übertragsmarke »Carry-Flag«

Syntax: CLC

Der CLC-Befehl ist genaugenommen kein arithmetischer Befehl. Er ist lediglich für die Steuerung bestimmter arithmetischer Operationen von großer Bedeutung. Das Kommando löscht das Carry-Flag im Kennzeichenregister des Prozessors. Das Carry-Flag wird vom Prozessor immer dann gesetzt, wenn bei einer Operation ein arithmetischer Übertrag stattgefunden hat, wenn sich das Ergebnis also nicht mit den zur Verfügung stehenden Bits darstellen ließ. In einem solchen Fall wird das Carry-Flag als 17. oder 9. Bit des Ergebnisses betrachtet. Vor allem in bezug auf die ADC- und SBB-Befehle stellt das Löschen des Carry-Flags eine wichtige Funktion dar.
Beispiel:

```
MOV CX,WERT1 ;lade CX mit Speicherstelleninhalt
CLC ;lösche Carry-Flag
ADC CX,100 ;und addiere 100 (alternativ könnte hier auch nur der ADD-Befehl zum Einsatz kommen)
```

CMC
»Complement Carry Flag«, invertiere den Status des Carry-Flags

Syntax: CMC

Dieser Befehl »schaltet« das Carry-Flag um. Ist es Eins, wird es auf Null gesetzt, ist es Null, dann wird es auf Eins gesetzt. Vor allem im Zusammenhang mit der Addition und Subtraktion ist der Zustand des Carry-Flags für das Ergebnis von Bedeutung.
Beispiel:

```
CMC ;schalte Carry um
```

CMP
»Compare«, vergleiche zwei Zahlen

Syntax: CMP <Zieloperand>,<Quelloperand>

Dieser Befehl vergleicht den Zieloperanden mit dem Quelloperanden. Beide Werte können entweder im Byte- oder Wort-Format vorliegen. Um den Vergleich durchzuführen, wird der Quelloperand vom Zieloperanden subtrahiert. Im Gegensatz zum Subtraktionsbefehl wird das Ergebnis jedoch nicht gespeichert, sondern es werden nur die Prozessor-Flags entsprechend dem Ergebnis gesetzt. Meist folgt auf einen CMP-Befehl ein bedingter Sprung.
Beispiel:

```
CMP AX,100 ;ist AX = 100?
JA DRUEBER ;nein, AX ist größer (»above«)
JB DRUNTER ;nein, AX ist kleiner (»below«)
... ;ja, AX = 100
```

CWD
»Convert word to double word«, konvertiere eine 16- in eine 32-Bit-Zahl

Syntax: CWD

Ähnlich dem CBW-Kommando, das ein Byte in ein Wort umwandelt, ändert der CWD-Befehl das im AX-Register gespeicherte Wort in ein Doppelwort um, das er im Registerpaar DX:AX ablegt. Das höherwertige Wort des Doppelwortes steht dann im DX- und das niederwertige Wort im AX-Register. Wenn AX kleiner als 8000hex ist, dann erhält DX den Inhalt 0000, ansonsten FFFFhex. Der Inhalt von AX bleibt unverändert. Der Befehl ist daher nur bei der Arbeit mit vorzeichenbehafteten Binärzahlen sinnvoll, da hier das Vorzeichen von AX auch als Vorzeichen des Doppelwortes DX:AX erhalten bleibt.
Beispiel:

```
MOV AX,1240 ;lade 1240 nach AX
NEG AX ;bilde -1240
CWD ;und wandle in Doppelwort um
```

DAA
»Decimal adjust after addition«, korrigiere das Ergebnis der Addition zweier gepackter BCD-Zahlen

Syntax: DAA

Das DAA-Kommando wird im Zusammenhang mit gepackten BCD-Zahlen verwendet (siehe auch den Befehl »AAA«, der mit ungepackten Zahlen arbeitet). Es erlaubt, Rechenfehler beim Umgang mit BCD-Zahlen nachträglich zu justieren. Dazu wandelt der Befehl den Inhalt des AL-Re-

gisters zu einer gepackten BCD-Zahl (ebenfalls im AL-Register) um.
Beispiel:

```
MOV AL,45h ;AL mit BCD-Zahl 45 laden
MOV CL,17h ;CL mit BCD-Zahl 17 laden
ADD AL,CL ;beide Zahlen addieren
DAA ;und wieder in eine BCD-Zahl wandeln
```

DAS

»Decimal adjust after subtraction«, korrigiere das Ergebnis der Subtraktion zweier gepackter BCD-Zahlen

Syntax: DAS

Dieses Kommando justiert das Ergebnis einer Subtraktion von zwei gepackten BCD-Zahlen (siehe auch den Befehl »AAS«, der mit ungepackten Zahlen arbeitet). Da der Prozessor diese Zahlen wie Binärzahlen subtrahiert, können Fehler im Ergebnis auftreten. Der DAS-Befehl wandelt das binäre Ergebnis der Subtraktion zweier gepackter BCD-Zahlen in eine ebenfalls gepackte BCD-Zahl um.
Beispiel:

```
MOV AL,77h ;AL mit BCD-Zahl 77 laden
MOV BL,19h ;BL mit BCD-Zahl 19 laden
SUB AL,BL ;subtrahiere beide Werte
DAS ;und wieder in eine BCD-Zahl umwandeln.
```

DEC

»Decrement«, vermindere

Syntax: DEC <Zieloperand>

Der DEC-Befehl vermindert den Inhalt des Zieloperanden um Eins. Dabei kann es sich um ein Register (aber kein Segmentregister) oder eine Speicheradresse handeln. Der Zieloperand kann Byte- oder Wort-Format besitzen.
Der Befehl wird vor allem in Schleifen häufig benutzt, um einen Index auf eine Tabelle, einen Zähler oder ähnliches anzupassen. Im Gegensatz zur Subtraktion von Eins direkt mit dem SBB- oder SUB-Kommando verändert der DEC-Befehl weder das Carry- noch das Zero-Flag, außerdem ist er wesentlich schneller.
Beispiel:

```
DEC BX ;zähle Index herab
DEC TABELLE[BX] ;und den Wert an der entsprechenden Stelle einer Tabelle
```

DIV

»Divide«, dividiere

Syntax: DIV <Quelloperand>

Im Gegensatz zu manch anderen Prozessoren besitzen die Intel-Prozessoren eigene Divisionsbefehle. Zwar ist der Divisionsbefehl nicht sonderlich schnell (er benötigt bis zu 172 Taktzyklen), dafür aber sehr komfortabel. Mit dem DIV-Befehl wird der Wert im AX-Register durch den Quelloperanden dividiert. Dabei wird davon ausgegangen, daß es sich bei dem Dividenden und dem Divisor um vorzeichenlose Binärzahlen handelt. Der Quelloperand kann Byte- oder Wort-Format besitzen und darf eine Speicheradresse oder ein Register sein. Liegt der Quelloperand im 16-Bit-Format vor, dann steht das Ergebnis der Division im Registerpaar AX:DX. Das AX-Register enthält den Quotienten und das DX-Register den bei der Division aufgetretenen Rest. Wurde AX hingegen durch einen 8-Bit-Wert geteilt, so steht der Quotient im AL-Register und der Rest im AH-Register.
Zu beachten ist, daß keine Möglichkeit besteht, direkt durch einen unmittelbaren Wert zu dividieren. In einem solchen Fall müssen Sie den Divisor zuerst in ein Register oder eine Speicherstelle laden und dann dividieren.
Beispiel:

```
MOV AX,SCHLEIF_WERT ;lade den Zähler einer Schleife
MOV BL,10 ;bereite Division durch 10 vor
DIV BL ;dividiere
CMP AH,0 ;War der Rest Null?
JE GLATT ;wenn ja, dann verzweige zum Label GLATT
```

IDIV

»Integer divide«, dividiere zwei Integer-Zahlen unter Berücksichtigung des Vorzeichens

Syntax: IDIV <Quelloperand>

Dieser Befehl führt eine Division des AX-Registers durch »Quelloperand« unter Berücksichtigung des Vorzeichens durch. Dabei wird davon ausgegangen, daß es sich bei dem Dividenden und dem Divisor um vorzeichenbehaftete Binärzahlen handelt. Der »Quelloperand« kann Byte- oder Wort-Format besitzen und darf eine Speicheradresse oder ein Register darstellen. Hat der »Quelloperand« 16-Bit-Format, dann steht das Ergebnis der Division im Registerpaar AX:DX. Das AX-Register enthält den Quotienten und das DX-Register den bei der Division aufgetretenen Rest. Wurde hingegen durch einen 8-Bit-Wert geteilt, so steht der Quotient im AL-Register und der Rest im AH-Register.
Wie beim DIV-, so gibt es auch beim IDIV-Kommando keine Möglichkeit, direkt durch einen unmittelbaren Wert zu dividieren. In einem solchen Fall müssen Sie den Divisor zuerst in ein Register oder eine Speicherstelle laden, ehe Sie dividieren.
Beispiel:

```
MOV AX,SCHLEIF_WERT ;lade den Zähler einer Schleife
MOV BL,10 ;bereite Division durch 10 vor
IDIV BL ;dividiere
CMP AH,0 ;War der Rest null?
JE GLATT ;wenn ja, dann springe zum Label GLATT
```

IMUL

»Integer multiply«, multipliziere zwei Zahlen unter Berücksichtigung des Vorzeichens

Syntax: IMUL <Quelloperand>

Dieses Kommando multipliziert den Inhalt des AX-Registers mit dem Quelloperanden. Dabei wird davon ausgegangen, daß der Quelloperand (Register oder Speicheradresse) und das AX-Register eine vorzeichenbehaftete Binärzahl enthalten (zum Beispiel 80hex, entspricht -128). Es darf mit 8-Bit-Werten oder mit 16-Bit-Werten multipliziert werden. Im ersten Fall wird nur die untere Hälfte von AX, also AL, zur Multiplikation verwendet; das Ergebnis der Multiplikation steht dann im AX-Register. Im zweiten Fall wird das gesamte AX-Register als erster Faktor benutzt, das Ergebnis steht dann im Registerpaar AX:DX. Das höherwertige Wort findet sich dabei im DX-Register, das niederwertige im AX-Register.
Beim IMUL-Kommando besteht keine Möglichkeit, direkt mit einem Wert zu multiplizieren. In einem solchen Fall laden Sie den Faktor zuerst in ein Register oder eine Speicherstelle und multiplizieren dann.
Beispiel:

```
MOV AX,65 ;lade 65 nach AX
MOV DL,20 ;lade 20 nach DL
IMUL DL ;bilde AX x DL
```

INC

»Increment«, erhöhe um Eins

Syntax: INC <Zieloperand>

Der INC-Befehl erhöht den Zieloperanden um den Wert Eins. Bei Zieloperand kann es sich um eine Speicheradresse oder ein Register (aber kein Segmentregister) handeln; der Zieloperand kann Byte- oder Wort-Format haben.
Dieser Befehl wird vor allem in Schleifen häufig benutzt, um einen Index auf eine Tabelle, einen Zähler oder ähnliches anzupassen. Im Gegensatz zur Addition von Eins direkt mit dem ADD- oder ADC-Kommando verändert der INC-Befehl weder das Carry- noch das Zero-Flag und ist schneller.
Beispiel:

```
INC SI ;erhöhe Index
INC TABELLE[SI] ;erhöhe den Wert in einer Tabelle an der entsprechenden Stelle
```

MUL

»Multiply«, multipliziere

Syntax: MUL <Quelloperand>

Dieses Kommando führt eine vorzeichenlose Multiplikation des AX-Registers mit dem Quelloperanden durch, der Byte- oder Wort-Format besitzen und ein Register oder ein Speicherbezug sein darf.
Wenn der Quelloperand ein 16-Bit-Wert ist, dann steht das Produkt nach der Ausführung der Mulitplikation in DX:AX. Der höherwertige Teil des

Ergebnisses steht im DX- und der niederwertige Teil im AX-Register. Wird hingegen mit einem 8-Bit-Faktor multipliziert, so wird nur AL für die Multiplikation benutzt. Dann benötigt die Operation nur 16 Bit, die höherwertigen acht Bit im AH- und die niederwertigen acht Bit im AL-Register.
Möchten Sie das AX-Register mit einer Konstanten multiplizieren, dann muß diese zuerst in ein Register oder eine Speicherstelle übertragen und dann multipliziert werden.
Beispiel:

```
MOV AX,FAKTOR1 ;lade ersten Faktor
MOV BX,FAKTOR2 ;lade zweiten Faktor
MUL BX ;multipliziere beide
```

NEG
»Negate«, invertiere das Vorzeichen

```
Syntax: NEG <Zieloperand>
```

Der NEG-Befehl bildet das Zweierkomplement des Zieloperanden. Dieser darf entweder Byte- oder Wort-Format haben und ein Register oder eine direkte Speicheradresse angeben.
Mit dem Kommando wandeln Sie eine Zahl in eine negative Zahl um, wenn Sie mit vorzeichenbehafteten Binärzahlen arbeiten.
Beispiel:

```
;nachfolgende Sequenz simuliert den
nicht erlaubten Befehl SUB 1000,AX (un-
mittelbare Werte sind als Zieloperand
beim SUB-Befehl nicht erlaubt)
NEG AX ;wandle AL in eine negative Zahl
ADD AX,1000 ;und addiere 1000
```

SBB
»Subtract with borrow«, subtrahiere unter Berücksichtigung des Carry-Flags

```
Syntax: SBB <Zieloperand>,<Quell-
operand>
```

Dieser Befehl subtrahiert den Quelloperanden vom Zieloperanden und legt das Ergebnis wieder im »Zieloperanden« ab. Die beiden Operanden dürfen entweder Byte- oder Wort-Format besitzen. Das SBB-Kommando berücksichtigt einen möglichen Übertrag im Carry-Flag. Es wird also immer zusätzlich eine Eins subtrahiert, falls das Carry-Flag gesetzt ist, sonst nicht.
Die Berücksichtigung des Übertrags ist vor allem bei nachfolgenden Subtraktionen mehrerer Zahlen von Bedeutung. Sollen beispielsweise zwei 32-Bit-Zahlen voneinander abgezogen werden, so subtrahiert man zuerst die beiden höherwertigen Worte ohne Beachtung des Übertrags (also mit dem SUB-Befehl) oder mit gelöschtem Übertrag. Dann subtrahiert man die beiden niederwertigen Worte der Zahlen unter Berücksichtigung des Übertrags. In einem solchen Fall wird ein möglicher Übertrag aus der ersten Operation in das zweite Ergebnis übernommen und so ein korrektes Endergebnis erzielt.
Wichtig ist, daß mit dem SBB-Kommando nicht zwei Speicherinhalte unmittelbar voneinander abgezogen werden können. Statt dessen übertragen Sie eine der Zahlen zuerst in ein Register. Ähnliches gilt für das Abziehen eines Speicherinhalts oder eines Registers von einer Konstante.
Beispiel:

```
;Subtraktion zweier 32-Bit-Zahlen
MOV AX,HIGH_WORT1 ;lade höherwertiges
Wort1 nach AX
SUB AX,HIGH_WORT2 ;ziehe höherwertiges
Wort2 ab
MOV BX,LOW_WORT1 ;lade niederwertiges
Wort1 nach BX
SBB BX,LOW_WORT2 ;und ziehe niederwerti-
ges Wort2 unter Berücksichtigung des
Übertrags ab
```

STC
»Set carry«, setze die Übertragsmarke »Carry-Flag«

```
Syntax: STC
```

Der STC-Befehl setzt das Carry-Flag, mit dem der Prozessor bei arithmetischen Operationen einen Übertrag angezeigt. Daher ist der Zustand des Carry-Flags bei vielen arithmetischen Operationen von Interesse.
Beispiel:

```
STC ;setze Carry-Flag
```

SUB
»Subtract«, subtrahiere

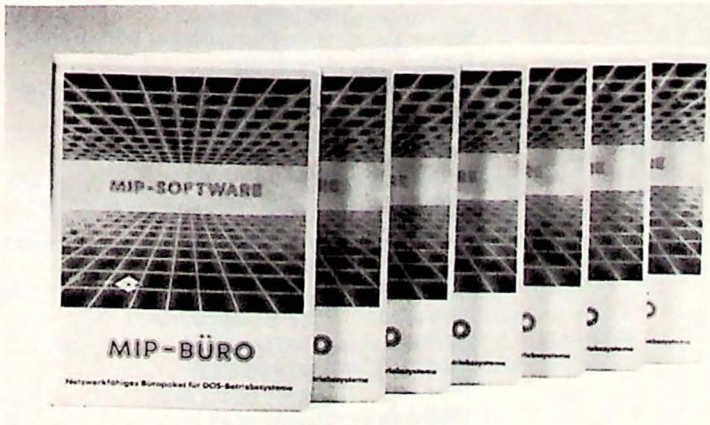
```
Syntax: SUB <Zieloperand>,<Quell-
operand>
```

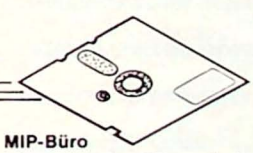
Mit diesem Befehl subtrahiert der Prozessor den Quelloperanden vom Zieloperanden und legt das Ergebnis wieder im Zieloperanden ab. Die beiden Operanden dürfen entweder Byte- oder Wort-Format besitzen. Das SUB-Kommando berücksichtigt einen möglichen Übertrag im Carry-Flag im Gegensatz zum SBB-Befehl nicht. Wichtig ist, daß Sie mit dem SUB-Kommando zwei Speicherinhalte nicht unmittelbar voneinander abziehen können. Statt dessen übertragen Sie eine der Zahlen zuerst in ein Register. Ähnliches gilt für das Abziehen eines Speicherinhalts oder eines Registers von einer Konstanten.
Beispiel:

```
MOV AH,100 ;lade 100 nach AH
SUB BL ;ziehe BL ab
```

(Martin Althaus/tr)

KAYSER DATEN-TECHNIK GmbH
Geseker Str. 43 · 4796 Salzkotten Telefon 05258/8022 Telefax 05258/4298





MIP-Büro
Internationales Büropaket mit Rechnungswesen, Faktura, Lager, Versand, Postvertrieb, Buchhaltung, Mahnwesen, offene Posten, Präsentationsgrafik, Stücklistenverwaltung, Textverarbeitung, Termine, Stammdaten, freie Steuersätze, Formulargenerator, frei wählbare Währung, netzwerkfähig,
1 Diskette 5,25 Zoll,
1 Diskette 3,5 Zoll,
Handbuch

299,–

MIP-Termplus
Netzwerkfähige Terminverwaltung mit reichhaltigen Auswertungen, Statistik, Grafik **69,–**

MIP-Barcode
Druckt beliebige Texte als Barcode, Seriendruck, Druckertreiber für IBM*, Epson* und HP-Laser* **69,–**

MIP-Clipper
50 Clipperfunktionen für den Profi, z.B. Passwortschutz, Backup, Restore, Kalender, Fenster usw. **169,–**

MIP-Textplus
Textverarbeitung und Programmeditor mit Zeilenumbruch, Blocksatz, Makros, Suchfunktion, Serienbrieffunktion; kompatibel zum MIP-Büro, -Fakt, -Post **69,–**

MIP-Kasse
Kassenbuch und Einnahme/Überschußrechnung, täglicher Kassenstand, freie Kontenverwaltung USt.-Listen **69,–**

MIP-Druckplus
Programmm zum Erstellen beliebiger Etiketten und Formulare in verschiedenen Schriften, speicherbar, Seriendruck **69,–**

MIP-Post
Postvertrieb mit Druck von Paket-, Nachnahme-, und Zahlkarten, Paket-Aufkleben, umfangreiche Adreßkartei, Notizen **69,–**

Händleranfragen erwünscht! **Demo 29,–**
*eingetragene Warenzeichen

Top Secret: das BIOS-Datensegment

Streng geheim scheint der Bereich des Arbeitsspeichers zu sein, in dem das BIOS seine Parameter ablegt, denn es ist sehr schwierig, über dieses Datensegment nähere Informationen zu erhalten. Dabei ist es sehr interessant, hierüber mehr zu wissen, denn durch Auslesen und Manipulieren der BIOS-Parameter läßt sich besonders trickreich programmieren.

Der 1 MByte große Speicherbereich des PC ist nach einem festen Schema unterteilt. Während die oberen 384 KByte des Speichers für Bildschirmspeicher, Betriebssystemerweiterungen und das BIOS (»Basic Input Output System«, enthält die grundlegenden Routinen zur Datenein- und Datenausgabe an die PC-Hardware) fest reserviert sind, bilden die unteren 640 KByte den Arbeitsspeicher, der die Programme und Daten des Anwenders, aber auch das Betriebssystem MS-DOS und die für den Betrieb des Computers notwendigen Parameter des BIOS-Datensegments enthält.

In der folgenden Tabelle finden Sie all die Informationen, die wir für Sie über das BIOS-Datensegment zusammentragen konnten: Speicheradressen, in denen zum Beispiel die verwendeten Schnittstellen, die aktuelle Systemkonfiguration, der Tastaturstatus, die aktive Grafikausrüstung oder Parameter zu den Disketten- und Festplattenlaufwerken abgelegt sind.

Sie können die Informationen aus der Tabelle beispielsweise dazu benutzen, um einem Programm die Abfrage der PC-Konfiguration zu erlauben, so daß es sich an jedes Computersystem individuell anpassen kann. Auch ist es durch Manipulieren der Speicherzellen möglich, sich über die Voreinstellungen des BIOS hinwegzusetzen und andere Parameter zu verwenden. Auf diese Weise können Sie zum Beispiel die Timeout-Zeiten für die verschiedenen PC-Schnittstellen neu einstellen.

Abschließend ein konkretes Beispiel: Möchten Sie beispielsweise die Datenausgaben auf die parallele Druckerschnittstelle von »lpt1« auf »lpt2« umleiten, so müssen Sie lediglich die Angaben über die Portadressen der beiden Schnittstellen miteinander vertauschen. In GW-Basic geschieht dies folgendermaßen:

```
10 DEF SEG = 0
20 lsbadr = PEEK(&H408)
30 msbadr = PEEK(&H409)
40 POKE &H408, PEEK(&H40A)
50 POKE &H409, PEEK(&H40B)
60 POKE &H40A, lsbadr
70 POKE &H40B, msbadr
80 DEF SEG
```

Das gleiche Beispiel unter Quick oder Turbo Pascal lautet:

```
AltAdr := MemW[$40:$08];
MemW[$40:$08] := MemW[$40:$0A];
MemW[$40:$0A] := AltAdr;
```

Natürlich läßt sich in diesem Fall dasselbe Ergebnis auch mit dem MS-DOS-Befehl

```
mode lpt1=lpt2
```

erzielen, doch gibt es für die wenigsten Tricks, die mit dem BIOS-Datensegment möglich sind, eine Entsprechung auf MS-DOS-Ebene.

(ma)

Adresse (hex)	Funktion
00000 – 003FF	Interrupt-Vektoren der Software-Interrupts 0 bis 255
00400 – 0040F	Portadressen der parallelen und seriellen PC-Schnittstellen (ist eine Schnittstelle nicht vorhanden, so lautet die Portadresse 0)
00400 – 00401	Portadresse der seriellen Schnittstelle »com1«
00402 – 00403	Portadresse der seriellen Schnittstelle »com2«
00404 – 00405	Portadresse der seriellen Schnittstelle »com3«
00406 – 00407	Portadresse der seriellen Schnittstelle »com4«
00408 – 00409	Portadresse der parallelen Schnittstelle »lpt1«
0040A – 0040B	Portadresse der parallelen Schnittstelle »lpt2«
0040C – 0040D	Portadresse der parallelen Schnittstelle »lpt3«
0040E – 0040F	Portadresse der parallelen Schnittstelle »lpt4«
00410	**Computersystem-Statusbyte** Bit 0 = 0: Kein Diskettenlaufwerk installiert Bit 0 = 1: Mindestens ein Diskettenlaufwerk installiert Bit 1 = 0: Arithmetikprozessor 8087 nicht vorhanden Bit 1 = 1: Arithmetikprozessor 8087 vorhanden Bit 2 und Bit 3: Speicher auf der Mutterplatine in 16-KByte-Einheiten (bei modernen PCs mit mehr als 64 KByte auf der Mutterplatine sind beide Bit gesetzt) Bit 5 = 0 und Bit 4 = 0: keine Grafik Bit 5 = 0 und Bit 4 = 1: 40x25 Zeichen (Farbgrafik) Bit 5 = 1 und Bit 4 = 0: 80x25 Zeichen (Farbgrafik) Bit 5 = 1 und Bit 4 = 1: 80x25 Zeichen (Monochromgrafik) Bit 6 und Bit 7: Anzahl der über eins hinausgehenden Diskettenlaufwerke
00411	**Schnittstellen-Statusbyte** Bit 0: Unbenutzt Bit 1 und 2: Anzahl der initialisierten seriellen Schnittstellen Bit 3: 0 (konstant) Bit 4 = 0: Kein Gameportadapter vorhanden Bit 4 = 1: Gameportadapter vorhanden Bit 5: Unbenutzt Bit 6 und 7: Anzahl der initialisierten parallelen Schnittstellen
00412	Test-Flag für Initialisierung
00413 – 00414	Größe des Arbeitsspeichers in KByte
00415 – 00416	Größe des Speichers auf Erweiterungskarten in KByte
00417	**1. Tastatur-Statusbyte** Bit 0 = 1: [Right Shift] gedrückt Bit 1 = 1: [Left Shift] gedrückt Bit 2 = 1: [Ctrl] gedrückt Bit 3 = 1: [Alt] gedrückt Bit 4 = 1: [Scroll Lock] aktiv Bit 5 = 1: [Num Lock] aktiv Bit 6 = 1: [Shift Lock] aktiv Bit 7 = 1: [Ins] aktiv

00418	**2. Tastatur-Statusbyte**
	Bit 0: [Left Ctrl] gedrückt Bit 1: [Left Alt] gedrückt Bit 2: [Sys Request] gedrückt Bit 3 = 1: [Pause] aktiv Bit 4 = 1: [Scroll Lock] gedrückt Bit 5 = 1: [Num Lock] gedrückt Bit 6 = 1: [Shift Lock] gedrückt Bit 7 = 1: [Ins] gedrückt
00419 – 0043D	**Tastaturpufferbereich**
00419	Aktuelles Zeichen, dessen ASCII-Wert mit der Alt-Taste in Ziffernform eingegeben wurde
0041A – 0041B	Offsetadresse des nächsten zu lesenden Zeichens im Tastaturpuffer (erforderliche Segmentadresse: 0040hex)
0041C – 0041D	Offsetadresse der nächsten zu beschreibenden Position im Tastaturpuffer (erforderliche Segmentadresse: 0040 hex) Der Tastaturpuffer ist leer, wenn die beiden zuvor aufgeführten Offsetadressen übereinstimmen.
0041E – 0043D	Tastaturpuffer (16 x 2 Byte, bestehend aus erweitertem ASCII-Wert und Tastennummer)
0043E – 00448	**Diskettenlaufwerkparameter**
0043E	Seek-Status der Diskettenlaufwerke
0043F	Motor-Status (gibt aktive Laufwerksmotoren an)
00440	Schleifenwert für Anlaufzeit der Diskettenmotoren
00441	Statusbyte der letzten Diskettenoperation (hex) 00: Kein Fehler aufgetreten 01: Ungültiger Befehl 02: Adreßmarke des angesprochenen Sektors nicht gefunden 03: Datenträger ist schreibgeschützt 04: Sektor nicht gefunden 08: Überlauf im DMA-Controller (Datenübertragung zu schnell) 09: Ungültiger Versuch, die 64-KByte-Grenze des DMA-Controller zu überschreiben 10: Ungültiger CRC (Prüfsummenfehler beim Lesen oder Verifizieren) 20: Fehler im Controller (zum Beispiel Spur 0 nicht gefunden) 40: Spur nicht gefunden 80: Laufwerk reagiert nicht (Wartezeit überschritten)
00442 – 00448	Status-Bytes des Disketten-Controller 765
00449 – 00466	**Grafikkartenparameter**
00449	Aktueller Videomodus (hex) 00 bis 06: CGA 07: Monochromadapter/Hercules 08 bis 0A: PC Junior 0D bis 10: EGA 11 bis 13: VGA
0044A – 0044B	Zeichen pro Zeile (Textmodus)
0044C – 0044D	Speicherbedarf einer Bildschirmseite in Byte
0044E – 0044F	Offsetadresse der aktiven Bildschirmseite relativ zum Anfang des Bildschirmspeichers
00450 – 0045F	Zwischenspeicher für Cursor-Positionen (Spalte/Zeile) der Bildschirmseiten 0 bis 7
00460	Start-Rasterzeile des Cursors (CGA: 0 bis 7, Monochromadapter/Hercules/EGA: 0 bis 13, VGA: 0 bis 15)
00461	End-Rasterzeile des Cursors (CGA: 0 bis 7, Monochromadapter/Hercules/EGA: 0 bis 13, VGA: 0 bis 15)
00462	Nummer der aktiven Bildschirmseite
00463	Portadresse des Video-Controller auf der aktiven Grafikkarte (Monochromadapter/Hercules: 3B4hex, CGA/EGA/VGA: 3D4hex)
00465	Inhalt des Modus-Auswahlregisters (Portadresse 3B8hex oder 3D8hex)
00466	Inhalt des Farb-Auswahlregisters (nur CGA/EGA/VGA: Portadresse 3D9hex)
00467 – 0046B	Reserviert (früher Pufferspeicher für Kassettenrecorder-Funktionen)
0046C – 00473	**Diverse Systeminformationen**
0046C – 0046F	32-Bit-Zähler der Systemuhr (Takt: 18,2 Hz)
00470	24-Stunden-Anzeige 0: Seit dem Start der Systemuhr sind noch keine 24 Stunden vergangen 255: Seit dem Start der Systemuhr sind über 24 Stunden vergangen
00471	Break-Byte 0: [Ctrl-C] beziehungsweise [Ctrl-Break] wurde nicht gedrückt 128: [Ctrl-C] beziehungsweise [Ctrl-Break] wurde gedrückt
00472 – 00473	Systemreset-Statuswort 1234: RAM-Test wird bei einem Reset durch [Ctrl-Alt-Del] ausgelassen (Warmstart) 4321: RAM-Test wird bei einem Reset durch [Ctrl-Alt-Del] nicht ausgelassen (Kaltstart)
00474 – 00477	**Festplattenlaufwerkparameter**
00474	Status-Byte der letzten Festplattenoperation (hex) 00: Kein Fehler aufgetreten 01: Ungültiger Befehl 02: Adreßmarke des angesprochenen Sektors nicht gefunden 03: Datenträger ist schreibgeschützt 04: Sektor nicht gefunden 05: Erfolgloser Reset-Versuch 07: Fehler bei der Initialisierung 08: Überlauf im DMA-Controller (Datenübertragung zu schnell) 09: Ungültiger Versuch, die 64-KByte-Grenze des DMA-Controller zu überschreiben 0D: Ungültige Sektorangabe beim Formatieren 10: Nicht korrigierbarer Lesefehler, Daten nicht gelesen) 11: Korrigierbarer Lesefehler, Daten gelesen 20: Fehler im Controller (zum Beispiel Spur 0 nicht gefunden) 40: Spur nicht gefunden 80: Laufwerk reagiert nicht (Wartezeit überschritten) AA: Festplattenlaufwerk nicht bereit BB: Fehlermeldung vom BIOS (zum Beispiel falscher Interleave-Wert) CC: Fehler beim Schreiben oder Formatieren aufgetreten FF: Nicht aufschlüsselbarer Fehler aufgetreten
00475	Anzahl der angeschlossenen Festplattenlaufwerke
00476	Kontroll-Byte (entspricht Byte 8 der Festplatten-Parametertabelle)
00477	Offsetwert relativ zu Portadresse 320hex (nur PC)

00478 – 0047F	Timeout-Zähler für die PC-Schnittstellen (geben in Einheiten zu etwa einer Sekunde (PC mit 4,77 MHz) beziehungsweise einer halben Sekunde (AT mit 6 MHz) die Zeit an, die der Computer auf eine Antwort des jeweils angeschlossenen Peripheriegerätes wartet)
00478	Timeout-Zähler für »lpt1«
00479	Timeout-Zähler für »lpt2«
0047A	Timeout-Zähler für »lpt3«
0047B	Timeout-Zähler für »lpt4«
0047C	Timeout-Zähler für »com1«
0047D	Timeout-Zähler für »com2«
0047E	Timeout-Zähler für »com3«
0047F	Timeout-Zähler für »com4«
00480 – 00481	Offsetwert der Startadresse des Tastaturpuffers (erforderliche Segmentadresse: 0040hex)
00482 – 00483	Offsetwert der Endadresse des Tastaturpuffers (erforderliche Segmentadresse: 0040hex)
00484 – 0048A	**Grafikausrüstung**
00484	Textzeilen pro Bildschirm - 1
00485	Höhe der Zeichenmatrix in Bildpunkten 08: CGA 14: Monochromadapter, Hercules, EGA 16: VGA
00487 – 0048A 00488	Videostatus-Bytes 248: CGA-Monitor (handelsüblicher Farbmonitor) 249: EGA-Monitor 251: Monochrom-Monitor
0048A	0: Keine VGA-Karte vorhanden 1 bis 255: VGA-Karte vorhanden
0048B – 0048E	**Weitere Festplattenlaufwerkparameter (nur AT)**
0048F – 00495	**Weitere Diskettenlaufwerkparameter (nur AT)**
00490	Status für Laufwerk A Bit 2 = 0, Bit 1 = 0 und Bit 0 = 0: 360-KByte-Diskette in 40-Spur-Laufwerk, nicht geprüft Bit 2 = 0, Bit 1 = 0 und Bit 0 = 1: 360-KByte-Diskette in Multifunktionslaufwerk, nicht geprüft Bit 2 = 0, Bit 1 = 1 und Bit 0 = 0: 1,2-MByte-Diskette in Multifunktionslaufwerk, nicht geprüft Bit 2 = 0, Bit 1 = 1 und Bit 0 = 1: 360-KByte-Diskette in 40-Spur-Laufwerk, geprüft Bit 2 = 1, Bit 1 = 0 und Bit 0 = 0: 360-KByte-Diskette in Multifunktionslaufwerk, geprüft Bit 2 = 1, Bit 1 = 0 und Bit 0 = 1: 1,2-MByte-Diskette in Multifunktionslaufwerk, geprüft Bit 2 = 1, Bit 1 = 1 und Bit 0 = 1: Aktueller Laufwerkszustand unbekannt Bit 3: Reserviert Bit 4 = 1: Disketten- und Laufwerkstyp festgestellt Bit 5 = 1: »Double Step« aktiv (40-Spur-Modus bei Multifunktionslaufwerk) Bit 7 = 0 und Bit 6 = 0: 500 KBit/s Datenübertragungsrate Bit 7 = 0 und Bit 6 = 1: 300 KBit/s Datenübertragungsrate Bit 7 = 1 und Bit 6 = 0: 250 KBit/s Datenübertragungsrate Bit 7 = 1 und Bit 6 = 1: Reserviert
00491	Status für Laufwerk B (Bedeutung der Bits, siehe oben)
00494	Letzte Spur von Laufwerk A
00495	Letzte Spur von Laufwerk B
00496 – 00497	Weitere Tastaturparameter (nur AT)
00498 – 004A0	Echtzeituhrparameter (nur AT)
004A1 – 004A7	Netzwerkparameter
004A8 – 004EF	Weitere Parameter (nur AT oder PS/2)
004F0 – 004FF	Von MS-DOS für den Anwender reserviert
00500 – 005FF	MS-DOS-, Basic- und Diskettenparameterbereich
00500	Statusbyte für Hardcopy-Funktion 0: Hardcopy korrekt beendet 1: Hardcopy in Bearbeitung 255: Fehler aufgetreten
00504	Status-Byte für den Betrieb mit nur einem Diskettenlaufwerk 0: Laufwerk wird gegenwärtig als »A« verwendet 1: Laufwerk wird gegenwärtig als »B« verwendet
0050F	Basic-Flag 0: Basic nicht geladen 1: Basic geladen
00510 – 00511	Segmentadresse von GW-Basic
00512 – 00513	Segmentadresse für Systemuhr-Interruptroutine (GW-Basic)
00514 – 00515	Offsetadresse für Systemuhr-Interruptroutine (GW-Basic)
00516 – 00517	Segmentadresse für Break-Tasten-Interruptroutine (GW-Basic)
00518 – 00519	Offsetadresse für Break-Tasten-Interruptroutine (GW-Basic)
0051A – 0051B	Segmentadresse für Diskettenfehler-Serviceroutine (GW-Basic)
0051C – 0051D	Offsetadresse für Diskettenfehler-Serviceroutine (GW-Basic)
0051E – 0051F	Reserviert
00522 – 0052C	Diskettenparametertabelle
00700	Start der speicherresidenten MS-DOS-Datei »io.sys«

Bei allen Adressen ist in der ersten Speicherstelle das niederwertige und in der zweiten Stelle das höherwertige Byte abgelegt.

Tabelle. Die Speicherbelegung des BIOS-Datensegments und der angrenzenden Speicherzellen

Bücher zur Assembler-programmierung

Diese DOS-Extra-Ausgabe enthält zwar eine Unmenge von Informationen rund um das Thema Assembler, doch kann sie natürlich nicht alle Fragen erschöpfend beantworten. Aus diesem Grund stellen wir Ihnen im folgenden weiterführende Literatur vor.

Die Titel, die Sie hier vorfinden, stellen nur eine kleine Auswahl aus dem gesamten Marktangebot dar. Viele Bücher beschäftigen sich nur mit einem bestimmten Prozessor, legen den Schwerpunkt auf einzelne Peripheriebausteine oder setzen sich lediglich mit einem bestimmten Assembler auseinander. Natürlich gibt es neben allgemeinen Werken auch Nachschlagewerke, Bücher für Einsteiger und Fortgeschrittene. Wer mehr wissen will, als in seinem Handbuch oder in den Norton-Guides zu Assembler aufgeführt ist, der wird wohl gerne zu einem dieser Bücher greifen.

Die aufgeführten Bücher sind meist neueren Datums und nach Preisen geordnet.

(Gisbert van Elsbergen/al)

Autor	Buchtitel	Verlag	Jahr	Seiten	Bemerkungen	Preis in DM
Lernbücher für Einsteiger						
Peter Heiß	PC Assemblerkurs – Einführung in die Maschinensprache für die IBM-Modelle XT, AT, PS/2 und Kompatible	Heise	1988	326	Einführung in die Assemblerprogrammierung für Basic- oder Pascal-Programmierer	44,80
W. Link	Assemblerprogrammierung. Eine gründliche Einführung unter MS-DOS	Franzis	1987	169	Einführung in Assembler	48
Isa Brors	Maschinensprache des IBM PC/AT und Kompatible in der Praxis	Hüthig	1988	338	–	54
David J. Bradley	Programmieren in Assembler für die IBM Personalcomputer	Hanser/ Prentice Hall Intern.	1986	366	Einführung in die 8088/8087-Programmierung mit hardwarenaher Programmierung, Infos zum ROM-BIOS und Einbindung in Hochsprachen	58
Peter Monadjemi	MS Makroassembler	Addison-Wesley	1988	300	Einführung in die Benutzung und das Arbeiten mit dem Microsoft-Makroassembler 5.1, Codeview, etc.	68
Jürgen Elsing	MS-DOS-Assemblerprogrammierung	IWT	1988	350	praktische Anwendungen von MS-DOS-Aufrufen	68
C. Vieillefond	Programmierung des 80286	Sybex	1987	509	Mit Befehlssatz und vielen Programmbeispielen	68
Rainer Kunz	Assemblerprogrammierung des 80386	Addison-Wesley	1988	400	Einführung in die Befehle, Programmierung und Debugging sowie die Befehle des 80387	78
Murray Sargent III/Richard Shoemaker/Ernst H. K. Stelzer	Assemblersprache und Hardware des IBM PC/XT/AT	Addison-Wesley	1988	471	Einblick in die Assemblersprache und Elektronik des PC	78
Peter Norton/ J. Socha	Peter Norton's Assemblerbuch	Markt & Technik	1988	352	inklusive zweier Disketten	79
Bertram Wohak	8086/286 Assembler	IWT	1987	605	umfangreiches Lehrbuch	88
Klaus-Dieter Thies	PC/XT/AT Assemblerbuch. Alle Befehle und Makroassembler	Tewi	1988	641	Einführung in die Programmierung, Makroassembler ASM86, ASM286, MASM und Befehlssatz-Referenz	98
Bücher für Fortgeschrittene						
Leo J. Scanlon	8086/8088/80286 Assembly Language	Brady Books	1988	378	–	58
David C. Willen, Jeffry I. Krantz	IBM PC/XT Assemblerprogrammierung CPU 8088	Tewi	1985	–	alle Daten zur Assembleransteuerung von 6845, 8237, 8253, 8255, 8259 und 8288	66
P. Monadjemi	PC-Programmieren in Maschinensprache	Markt & Technik	1988	623	inklusive Diskette	69
William H. Murray/ Chris H. Pappas	80386/80286 Assemblerpogrammierung für Experten	McGraw-Hill	1987	530	–	75
Volker Sasse	Intel 80386 Systemprogrammierung	Markt & Technik	1989	239	–	79
Klaus-Dieter Thies	80386 Die innovativen 80286/80386 Architekturen	Tewi	1988	–	–	89
Jürgen Hegner	8086/8088 Assemblerprogrammierung	KRS Verlagsges.	1988	425	besitzt Handbuchcharakter	98
Jürgen Hegner	80286 Assemblerprogrammierung	KRS Verlagsges.	1988	424	besitzt Handbuchcharakter	98
John H. Crawford/Patrick P. Gelsinger	Programming the 80386	Sybex	1988	775	in englisch; von den Designern des 80386-Prozessors	56,20
Nachschlagewerke						
R. Erskine	Assemblerprogrammierung für die Prozessoren 8086 und 8088	Oldenbourg	1986	103	besonders geeignet zum schnellen Nachschlagen	24,80
Jürgen Hegner	8086/8088 Quick Reference Guide Assembler	KRS Verlagsges.	1988	–	–	30
Harly Hahn	Complete Guide IBM-PC/AT Assembly Language	Scott, Foresman Company	1987	568	–	59
Referenzbücher zur Systemprogrammierung						
Heinz Wolek	MS-DOS-Systemprogrammierung. Anleitung zum praktischen Gebrauch	Oldenbourg	1988	244	für Einsteiger und Fortgeschrittene	48
Thom Hogan	PC-Referenz für Programmierer	Systhema	1988	520	Tabellen zu MS-DOS, PC-DOS, MS-Windows, PC- und PS/2-Hardware	69

Assemblerprogramme in GW-Basic

Ein effektives Gespann

Die Vorteile von Basic liegen auf der Hand: Es läßt sich schnell erlernen, erlaubt kurze Programmentwicklungszeiten und ist fast für jeden PC-Besitzer verfügbar. Dennoch gibt es Problemstellungen, bei denen Sie mit reinem GW-Basic nicht auskommen. Vor allem bei geschwindigkeitsabhängigen Aufgabenstellungen oder bei effektiven Routinen für die Zeichenkettenverarbeitung muß GW-Basic meist passen.

Wenn Sie in GW-Basic programmieren, wird es Ihnen ab und zu passieren, daß geschwindigkeitsabhängige Routinen einfach zu lange dauern. In solch einem Fall ist es sinnvoll, kurze Programmteile vollständig oder teilweise in Maschinensprache zu formulieren. In GW-Basic sind dazu unterschiedliche Ansätze möglich.
Sie reservieren entweder oberhalb des Basic-Speichers Platz für Maschinenspracheprogramme und schreiben diese Programme mit »poke« in den Speicher, oder Sie laden diese Programme mit dem Bload-Befehl in den Speicher. Andererseits können Sie kurze Maschinenspracheprogramme direkt im Variablenspeicher von Basic ablegen und wie »normale« Variablen verwalten.
Es gibt einige typische Anwendungsbereiche für Maschinensprache unter GW-Basic. Ein wesentliches Teilgebiet ist die Realisierung von schnellen Routinen für die Bildschirmausgabe, ein anderes ist die schnelle Verarbeitung von Strings (Zeichenketten). Aufgrund der Tatsache, daß Basic bei jeder String-Zuweisung eine neue Zeichenkette anlegt und der Speicher daher von Zeit zu Zeit aufgeräumt werden muß, um die »String-Leichen« zu entfernen, ist Zeichenketten-Programmierung in Basic nicht nur sehr ineffizient, sondern auch sehr langsam.
Nehmen Sie einmal an, Sie entwickeln eine Funktion, die einen gegebenen String »umkehrt« und wieder im String speichert. Aus »Walter« würde dann »retlaW«, aus »der erste String« würde »gnirtS etsre red« und so weiter. In GW-Basic schreiben Sie ein solches Unterprogramm so:

```
10 umkehr$=""
20 for i=1 to len(zkette$)
30 umkehr$=umkehr$+mid$(zkette$,
len(zkette$)-i+1,1)
40 next i
50 zkette$=umkehr$
```

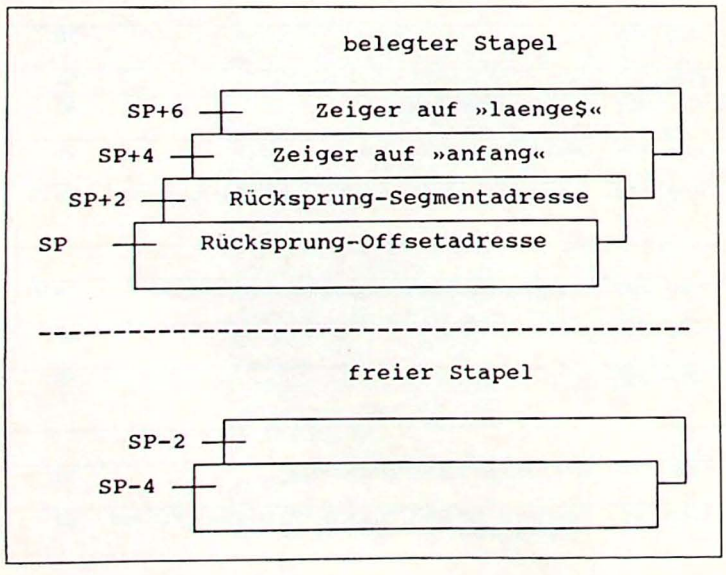
Dabei wird in der Variablen »zkette$« die umzukehrende Zeichenkette übergeben. Es wird Zeichen für Zeichen umgedreht und das Ergebnis wieder in der Variablen »zkette$« abgelegt.
Ein derartiges Programm arbeitet sehr langsam und uneffektiv. Wird das Programm beispielsweise mit einem String von zehn Zeichen Länge aufgerufen, werden insgesamt elf »String-Leichen« produziert. Allgemein gesagt werden überflüssige Strings in einer Anzahl von Stringlänge plus 1 bei jedem Aufruf des Unterprogramms erzeugt. Sie können sich schnell ausrechnen, wie lange das gutgeht, wenn zum Beispiel in einer Schleife 500 Strings mit jeweils 150 Zeichen umgewandelt werden sollen.

Häufig werden in GW-Basic bei Stringumkehrungen sogenannte »String-Leichen« produziert

Es ist daher sinnvoll, die gleiche Routine in Assembler zu programmieren. Dabei stellt sich zunächst die Frage, wie wird ein Assemblerprogramm unter GW-Basic aufgerufen? Dazu wird der Befehl »call« benutzt. Dem Call-Befehl wird als erster Parameter eine Adresse übergeben, die auf den Start der aufzurufenden Assemblerroutine weist. Innerhalb des mit der Anweisung »def seg« festgelegten Segments wird die Adresse als Offset auf ein Maschinenspracheprogramm interpretiert. Die Anweisung »far call« des Prozessors ruft dann das Maschinenspracheprogramm auf. Daher muß das Maschinenspracheprogramm auch als Far-Prozedur deklariert sein. Das folgende Gerüst läßt sich dazu verwenden:

```
code            segment
                assume cs:code
programm        proc far
;
;hierhin gehört Ihr Programm
;
programm        endp
code            ends
                end
```

Als weitere Parameter können Sie dem Maschinenspracheprogramm Basic-Variablen übergeben. Die im Call-Aufruf angegebenen Parameter werden »von links nach rechts« auf den Stack (oder zu deutsch Stapel) geschoben. Dabei werden keine unmittelbaren Werte (»call by value«), sondern vielmehr Zeiger auf die tatsächlichen Wertebereiche im Basic-Speicher (»call by reference«) übergeben.
Bei der Ausführung eines Call-Befehls geschieht nun folgendes: Zuerst



Der Zustand des Stacks nach einem Unterprogrammaufruf mit dem Call-Befehl

werden die Adressen der in »call« angegebenen Parameter auf den Stack geschoben. Dann wird das durch den ersten Parameter der Call-Anweisung ausgewählte Unterprogramm innerhalb des mit »def seg« gewählten Segments ausgeführt. Diese Anweisung erzeugt einen Far-Call-Maschinenbefehl des Prozessors. Das hat zur Folge, daß die Rücksprungadresse in der Reihenfolge Segment, Offset ebenfalls auf den Stack geschoben wird. Da nun aber der Stack von »oben« nach »unten« wächst, müssen die Werte in der umgekehrten Reihenfolge innerhalb eines Maschinenspracheprogramms wieder vom Stack gelesen werden.

Betrachten Sie dazu ein Beispiel. Wenn ein Maschinenspracheprogramm an der Adresse »startadr« aufgerufen und ihm die GW-Basic-Variablen »laenge$« und »anfang« als Parameter übergeben werden sollen, geschieht das folgendermaßen:

```
10 call startadr(laenge$,anfang)
```

Führt GW-Basic diesen Befehl aus, werden zunächst die Parameter der Reihe nach auf den Stack geschoben. Dazu benutzt GW-Basic die Offsetadressen der Variablen im Datensegment. GW-Basic schiebt also ein Wort für die Offsetadresse von »laenge$« und eines für die Offsetadresse von »anfang« auf den Stack. Danach löst es einen Far-Call-Maschinenbefehl aus. Dieses Kommando bewirkt, daß auch die Rücksprungadresse in das Basic-Programm mit zwei Worten auf dem Stack gespeichert wird. Erst dann wird zur Offsetadresse »startadr« verzweigt. Nach dem obigen Aufruf sieht der Stack aus, wie im Bild dargestellt.

Über das BP-Register sichern Sie die Rücksprungadresse auf den Stack

Möchten Sie nun die Variablen innerhalb eines Maschinenspracheprogramms auslesen oder verändern, so ist es nicht empfehlenswert, sie mit dem Pop-Befehl des Prozessors vom Stack zu lesen. Denn dann müssen Sie zuerst die Rücksprungadresse sichern und vor Verlassen des Unterprogramms wieder auf den Stack bringen. Einfacher ist es, den Stack direkt zu adressieren. Dazu verwenden Sie das BP-Register, indem Sie es mit dem Wert von SP initialisieren. Da aber das BP-Register in einem Assemblerprogramm unter GW-Basic nicht verändert werden darf, müssen Sie es vorher auf den Stack retten. Danach adressieren Sie die einzelnen Parameter auf dem Stack. Hierfür eignet sich folgende Sequenz:

```
push bp          ;Register BP retten
mov bp,sp        ;als Zeiger auf Stack
                  verwenden
```

Sie wird an der gekennzeichneten Stelle in das oben beschriebene Programmgerüst eingefügt. Nun sprechen Sie die einzelnen Variablen über das BP-Register an. Zu den in Abbildung 1 gezeigten Offsets der Variablen auf dem Stack muß nun natürlich noch der Wert 2 addiert werden, weil das eigene Programm für das Retten von BP ein weiteres Wort auf den Stack ablegt und der Stackzeiger SP daher um zwei vermindert wird. Am einfachsten ist es, die Variablen symbolisch zu adressieren. Sie erreichen das in Assembler durch die folgenden Zeilen:

```
laenge equ word ptr [bp+8]
anfang equ word ptr [bp+6]
```

Nun verwenden Sie innerhalb des Assemblerprogramms die symbolischen Namen »laenge« und »anfang« und haben damit automatisch Zugriff auf die Variablenzeiger auf dem Stack. Dabei beachten Sie unbedingt, daß es sich um Zeiger handelt. Um den tatsächlichen Wert der Variablen zu ermitteln, müssen Sie den Zeiger auslesen und als Offset im GW-Basic-Datensegment verwenden. Soll also zum Beispiel der Inhalt der Variablen »anfang« gelesen werden, sind dazu zwei Schritte notwendig:

```
mov si,laenge  ;lade den Zeiger auf »an-
                fang« nach SI
mov ax,[si]    ;lade den Inhalt von »an-
                fang» nach AX
```

Kennen Sie erst einmal die oben beschriebene Vorgehensweise des Basic-Interpreters beim Aufruf von Maschinenspracheprogrammen, ist es ein leichtes, eigene Assemblerprogramme zu formulieren. Deshalb zurück zum Beispiel. Es sollte eine Funktion entwickelt werden, die alle Zeichen eines Strings umdreht und das Ergebnis wieder im String speichert; die Funktion soll »Spiegel« heißen. Die grundsätzliche Vorgehensweise sieht folgendermaßen aus: Sie beginnen beim letzten Zeichen des Strings und speichern das als erstes Zeichen in einem Puffer ab. Dann übertragen Sie das vorletzte Zeichen des Strings als zweites Zeichen in den Puffer und so weiter. Sie benutzen also zwei »gegeneinander« laufende Indizes. Haben Sie auf diese Weise alle Zeichen des Strings übertragen, liegt der String im Puffer schon in der umgedrehten Form vor. Sie übertragen ihn dann nur noch in den alten String-Bereich.

Maschinenprogramme bis zu einer Länge von 255 Zeichen bringen Sie im Variablenspeicher von GW-Basic unter

Das Programm selbst ist ziemlich kurz. Es soll deshalb in einem String im Basic-Datenbereich untergebracht werden. Dadurch läuft es mit jeder Basic-Version, ohne daß erst Platz für ein Maschinenspracheprogramm oberhalb des Basic-Speichers reserviert werden muß. Alle Maschinenspracheprogramme bis maximal 255 Zeichen lassen sich auf diese Weise im normalen Variablenspeicher von Basic unterbringen. Längere Programme sind mit dieser Methode nicht adressierbar, da GW-Basic für die Länge einer String-Variablen maximal 255 Zeichen vorsieht.

Damit taucht aber schon das erste Problem bei der Realisierung der Spiegel-Routine auf. Wenn Strings bis zu 255 Zeichen lang sein dürfen, benötigen Sie zum Umdrehen des Strings auch einen Pufferspeicher von 255 Byte. Dann ist aber kein Platz mehr für das Maschinenspracheprogramm vorhanden. Dieses Problem lösen Sie dadurch, daß Sie dem Maschinenspracheprogramm zwei Parameter übergeben. Der erste Parameter ist der umzudrehende String und der zweite Parameter ein Leer-String von 255 Zeichen Länge, den

Byte	Bedeutung
00hex	Bload-Kennung FDhex
01hex	Low-Byte der Segmentadresse
02hex	High-Byte der Segmentadresse
03hex	Low-Byte der Offset-Adresse
04hex	High-Byte der Offset-Adresse
05hex	Low-Byte der Länge des Assemblerprogramms
06hex	High-Byte der Länge der Assemblerprogramms

Tabelle. Der Aufbau eines Bload-Headers

das Programm als Puffer verwendet. Diesen String legen Sie mit der Anweisung

```
puffer$=string$(255," ")
```

problemlos an. Im Maschinenspracheprogramm lesen Sie dann die Adresse des Puffer-Strings aus und benutzen ihn als Datenpuffer. Allerdings ist der Zugriff auf den Datenbereich eines Strings ein wenig schwieriger als der Zugriff auf den Datenbereich einer numerischen Variablen. Der Zeiger, der von GW-Basic auf den Stack geschoben wird, ist nämlich nicht, wie bei numerischen Variablen, ein Zeiger auf den String-Datenbereich, sondern ein Zeiger auf den String-Descriptor. Dieser Descriptor ist ein Datenblock, der Informationen über die Länge und die Position des Strings enthält. Als erstes Byte wird im String-Descriptor die Länge des Strings und im Wort dahinter der Offset des Strings im Basic-Datensegment gespeichert. Um also die Adresse des Puffer-Strings in das DI-Register zu laden, verwenden Sie die folgenden Befehle:

```
puffer equ word ptr [bp+6]
       mov si,puffer
       mov si,[si+1]
```

Die Definition »equ« zu Beginn dient wiederum nur der einfacheren Adressierbarkeit der Variablen. Sie können die Anweisung »word ptr [bp + 6]« auch direkt in den Quelltext einsetzen. Der Offset von sechs zum BP-Register setzt, wie oben beschrieben, voraus, daß in der Prozedur zuerst das BP-Register auf dem Stack gesichert wurde.

Haben Sie erst einmal die Adressen der Strings und die Länge des umzudrehenden Strings in drei Register geladen, so ist das weitere Vorgehen einfach. Sie addieren zunächst die Stringlänge zum Zeiger auf den umzudrehenden String und subtrahieren eins vom Ergebnis. Dadurch zeigt das Register dann auf das letzte Zeichen des Strings. Nun lesen Sie der Reihe nach ein Zeichen aus dem Quell-String und speichern es im Puffer. Der Zeiger auf den Quell-String wird nach der Operation dekrementiert, der Zeiger auf den Zielstring inkrementiert. Nun verringern Sie die String-Länge um eins.

Diesen Vorgang wiederholen Sie so lange, bis die String-Länge den Wert Null erreicht, also alle Zeichen des Strings verarbeitet sind. Dann ist der String umgedreht und muß nur noch vom Puffer wieder in den Datenbereich des Quell-Strings übertragen werden. Das passende Assemblerlisting »spiegel.asm« zu diesem Problem sehen sie in Listing 1.

Dieses Programm assemblieren Sie mit einem Assembler, wie etwa dem Microsoft-Assembler MASM oder dem Turbo-Assembler.

Alle Maschinenbefehle tragen Sie nun als Data-Zeilen in ein GW-Basic-Programm ein. In einer Schleife zu Beginn des GW-Basic-Programms lesen Sie dann die einzelnen Maschinenbefehle aus und konstruieren daraus einen String. Möchten Sie das Maschinenspracheprogramm einmal aufrufen, berechnen Sie in einem Unterprogramm zuerst die Adresse des Strings und springen dann in das Programm. Die Adreßberechnung muß dabei direkt vor dem Aufruf erfolgen, da sich der String im Zuge einer »Garbage Collection« im Speicher verschieben kann.

Assemblerprogramme lassen sich »getarnt« in GW-Basic-Programme einbinden

Mit dem oben beschriebenen Verfahren binden Sie das Programm »spiegel.asm« schnell in ein GW-Basic-Programm ein. Als Demonstration dazu dient Listing 2 (»spiegel.bas«), in dem die Maschinenroutine zum Umdrehen eines Strings aufgerufen wird. Probieren Sie hier einmal den Text »ein neger mit gazelle zagt im regen nie«, Sie werden sich wundern, welches Resultat Sie erhalten.

Die so ausführlich beschriebene Möglichkeit, Assemblerprogramme als Variablen »getarnt« in ein Basic-Programm einzubauen und aufzurufen, greift wie erwähnt nur für Programme bis zu maximal 255 Byte Länge. Wollen Sie längere Programme einbauen, empfiehlt es sich, GW-Basic mit dem Schalter »/M:54000« anzuweisen, am oberen Ende des Basic-Segments Platz freizuhalten. Sie schreiben dann ein Maschinenspracheprogramm mit dem Poke-Befehl in diesen Bereich. Der Aufruf des Programms und die Übergabe der Parameter geschieht ebenfalls wie oben beschrieben.

Unter GW-Basic können Sie auch externe Assemblerprogramme, die als Dateien vorliegen, einbinden

Als dritte Möglichkeit laden Sie ein auf der Diskette bereits als Com-Datei vorliegendes Assemblerprogramm mit dem Befehl »bload« in den Basic-Speicher und führen es aus. Voraussetzung dafür ist ebenfalls, daß oberhalb des Basic-Bereichs Platz für das Programm freigehalten wurde. Vorher müssen Sie allerdings das Com-Programm um sieben Byte erweitern, da Basic bei jedem Programm, das mit »bload« geladen werden soll, einen speziellen Header zu Beginn des Programms erwartet. Der Aufbau dieses Headers ist in der Tabelle dargestellt. Vom Programmierer muß nur das Byte 0 und das letzte Wort mit einem Wert versehen werden – alles andere trägt GW-Basic nach dem Laden ein.

Nun können Sie sich daranmachen, Ihre Basic-Listings durch Assemblerroutinen schneller, leistungsfähiger und effektiver zu programmieren. Denn dann lassen sich auch unter GW-Basic maschinennahe und flinke Programme entwickeln.

(Martin Althaus/wn)

```
[VC4]    1: ;Aufruf der Funktion mit:
[FT6]    2: ; CALL <Adresse>,<String>,<Puffer-String>
[657]    3: ;Übergeben einer Stringvariable <String>
[TU3]    4: ;und ein auf 255 Zeichen
[PH5]    5: ;initialisierter String, der als
[UY2]    6: ;Puffer arbeitet.
[WU6]    7: ;(c) Redaktionsbüro Everts und Hagedorn
[P3Ø]    8: ;
[SD2]    9: code       segment
[EA2]   1Ø:            assume cs:code
[P3Ø]   11: ;
[DX4]   12: ;symbolische Variablennamen
[P3Ø]   13: ;
[9E4]   14: string     equ word ptr [bp+8]
[ØD4]   15: puffer     equ word ptr [bp+6]
[P3Ø]   16: ;
[XR6]   17:      ;wird von Basic mit Far Call aufgerufen
```

```
[253]   18: programm  proc   far
[933]   19:      ;rette BP auf Stack
[N91]   2Ø:           push   bp
[P76]   21:      ;verwende BP als Zeiger auf den Stack
[1T1]   22:           mov    bp,sp
[XR5]   23:      ;lade Adr. des Puffer-Strings nach SI
[1F2]   24:           mov    si,puffer
[DB5]   25:      ;nun zeigt SI auf Stringadresse
[VB2]   26:           mov    si,[si+1]
[BW2]   27:      ;rette die Adresse
[S91]   28:           push   si
[EE5]   29:      ;initialisiere ES über den Stack
[W91]   3Ø:           push   cs
[R73]   31:      ;mit dem Wert von CS
[S71]   32:           pop    es
[7W5]   33:      ;lade Zeiger auf den String nach DI
[RG2]   34:           mov    di,string
```

```
[EX4]    35:       ;lese Länge des Strings aus
[R62]    36:             mov     cl,[di]
[ØC4]    37:       ;CX hat nun String-Länge
[GP1]    38:             xor     ch,ch
[DX3]    39:       ;sichere String-Länge
[6A1]    4Ø:             push    cx
[VB5]    41:       ;DI zeigt nun auf String-Adresse
[XA2]    42:             mov     di,[di+1]
[KV3]    43:       ;sichere diesen Zeiger
[991]    44:             push    di
[VR4]    45:       ;beginne bei letztem Zeichen
[WK1]    46:             add     di,cx
[1G5]    47:       ;nun zeigt DI auf letztes Zeichen
[T31]    48:             dec     di
[LS2]    49:       ;lese Zeichen aus
[2S3]    5Ø: schleife: mov     al,[di]
[BM4]    51:       ;und speichere es im Puffer
[182]    52:             mov     [si],al
[GB3]    53:       ;erhöhe Zielzeiger
[D51]    54:             inc     si
[C94]    55:       ;vermindere Quellzeiger
[T31]    56:             dec     di
[MD4]    57:       ;und lese nächstes Zeichen
[FE2]    58:             loop    schleife
[8S4]    59:       ;hole Stringadresse zurück
[371]    6Ø:             pop     di
[UT4]    61:       ;hole String-Länge zurück
[Ø81]    62:             pop     cx
[3P4]    63:       ;hole Pufferadresse zurück
[K71]    64:             pop     si
[DS3]    65:       ;vorwärts speichern
[ULØ]    66:             cld
[ØP3]    67:       ;übertrage den String
[372]    68: rep         movsb
[CX4]    69:       ;und hole BP wieder vom Stack
[F71]    7Ø:             pop     bp
[CB4]    71:       ;hole Parameter vom Stack
[R11]    72:             ret     4
[DF3]    73:       ;und beende Programm
[ØØØ]    74:
[354]    75:       ;hier endet die Prozedur
[VB2]    76: programm  endp
[2T2]    77:       ;hier das Segment
[BW1]    78: code      ends
[8F3]    79:       ;und hier das Listing
[4MØ]    8Ø:             end
```

Listing 1. »spiegel.asm« dreht die Reihenfolge der Zeichen eines Strings um

```
[687] 1ØØ '(c) Redaktionsbüro Everts und Hagedorn
[3HØ] 11Ø '
[LW1] 12Ø RESTORE
[DK2] 13Ø CHECKSUMME=Ø
[SØ2] 14Ø READ CODE$
[MX3] 15Ø IF CODE$="ENDE" THEN 19Ø
[KT6] 16Ø PROGRAMM$=PROGRAMM$+CHR$(VAL("&h"+CODE$))
[3C6] 17Ø CHECKSUMME=CHECKSUMME+VAL("&h"+CODE$)
[RP1] 18Ø GOTO 14Ø
[B93] 19Ø READ PRUEFSUMME$
[KU5] 2ØØ PRUEFSUMME=VAL("&h"+PRUEFSUMME$)
[81B] 21Ø IF CHECKSUMME<>PRUEFSUMME THEN PRINT "Fehler
      in den Datas!":STOP
[SY3] 22Ø PUFFER$=STRING$(255," ")
[BHØ] 23Ø '
[AX3] 24Ø 'hier Hauptprogramm
[HHØ] 25Ø '
[W41] 26Ø CLS
[1T6] 27Ø DEMO$="dieser text wird nun umgedreht"
[8B2] 28Ø PRINT DEMO$
[LU1] 29Ø GOSUB 38Ø
[KA2] 3ØØ PRINT DEMO$
[9C1] 31Ø PRINT
[T95] 32Ø PRINT "Probieren Sie es aus! "
[8L4] 33Ø INPUT "Umzudrehen: ",DEMO$
[6U1] 34Ø GOSUB 38Ø
[1B2] 35Ø PRINT DEMO$
[RP1] 36Ø GOTO 33Ø
[F31] 37Ø END
[RKA] 38Ø AUFRUF!=PEEK(VARPTR(PROGRAMM$)+2)*256+PEEK(VA
      RPTR(PROGRAMM$)+1)
[3U4] 39Ø CALL AUFRUF!(DEMO$,PUFFER$)
[YL1] 4ØØ RETURN
[Z18] 41Ø DATA 55,8b,ec,8b,76,Ø6,8b,74,Ø1,56,Øe,Ø7,8b,7
      e,Ø8,8a,Ød,32
[Z78] 42Ø DATA ed,51,8b,7d,Ø1,57,Ø3,f9,4f,8a,Ø5,88,Ø4,4
      6,4f,e2,f8,5f
[KS4] 43Ø DATA 59,5e,fc,f3,a4,5d,ca,Ø4,ØØ
[NY1] 44Ø DATA ENDE
[ZS1] 45Ø DATA 125f
```

Listing 2. Das GW-Basic-Programm »spiegel.bas« demonstriert das Einbinden von Assemblerroutinen

Externe 3,5"-/5,25"-Diskettenlaufwerke
auch für Ihren PC !!!

Geeignet für alle PC/XT/ATs! Das **externe Laufwerk** ist über eigens entwickelten Adapter anzuschließen. Vorhandene eingebaute Laufwerke können weiter benutzt werden. Umschaltung erfolgt am **externen Laufwerk** durch Tipptaster. Alle Anschlußkabel und Adapter werden mitgeliefert; geeignet für alle PC/XT/ATs mit DOS ab 3.2, **Amstrad** und **Schneider.**

Externe Laufwerke 3,5" ab **DM 369,-**
Externe Laufwerke 5,25" ab **DM 348,-**
Spezial-Controller 1,44 MB bzw. 1,2 MB/720-360 kB für XT u. AT **DM 168,-**

Externe Laufwerke auch für **IBM*** XT/AT u. PS/2, **Siemens** PCD/2, **Nixdorf** PC.

becos-Spezialanfertigungen:

Laufwerkkombinationen von Festplatte u. Laufwerk; z.B. 20 MB Festplatte und 360 kB/5,25"-Laufwerk.

Wir entwickeln u. fertigen Computer/-zubehör/-geräte auf Kundenwunsch!
Bitte rufen Sie uns an, oder schreiben Sie uns! Gern schicken wir Ihnen unser Lieferprogramm mit Preisliste.

IBM* = eingetr. Warenzeichen von International Business Machines Corp.

becos *Kriegenkamp 11*
4543 Lienen
Tel. 05483 / 10 34 · Fax 05483 / 14 54

Profi-Tools für QuickBASIC und Turbo BASIC

Schreiben Sie schnellere, leistungsfähigere und professionellere Programme! Wir helfen Ihnen dabei mit nützlichen Tools.

Zum Beispiel:
- **Toolboxen (Fenstertechnik, Menüs, DOS-Funktionen etc.)**
- **Relationale Datenbank mit komfortablem Masken-Editor**
- **Grafik-Paket (Geschäftsgrafik und Zeichensatz-Generator)**
- **Maus-Programmierung**
- **Laserdrucker-Unterstützung**

Alle Pakete mit ausführlich dokumentierten Quelltexten und Programmbeispielen. Wo erforderlich, kommen schnelle Assembler-Routinen zum Einsatz. Wollen Sie mehr aus Ihrem BASIC-Compiler herausholen? Wir informieren Sie gerne kostenlos!

Ingenieur-Büro Harald Zoschke
Berliner Str. 3
D-2306 Schönberg/Holstein
Telefon 0 43 44/61 66

Eingetr. Warenzeichen:
QuickBASIC: Microsoft; Turbo BASIC: Borland

PC-MultiKASSE

Ab jetzt haben Sie Ihre Belege voll im Griff; leicht, schnell, sicher und preiswert.
Belege aufwendig zu sortieren, können Sie vergessen!
Sortierte Listen auf »Knopfdruck«?
Selektieren nach verschiedensten Kriterien?
Nachträgliches Ändern, Löschen, Sortieren?

Mit PC-MultiKASSE kein Problem!

PCK-GRAF erstellt verschiedene Diagramme; Ausdrucke um 90 Grad gedreht auf DIN A4.

PC-MultiKASSE
inkl. PCKGRRAF ..... DM 248,–

Weitere Informationen? Schreiben Sie an:

M.M.W. MicroMarket-Worms
Joh.-Westermann-Platz 1,
D-4780 Lippstadt,
Telefon 0 29 41/5 92 90

Maschinen-Basic

Assembler in Quick Basic einbinden

Quick Basic sollte (zumindest dem Namen nach) eine schnelle Hochsprache sein. Doch manchmal treten auch hier Geschwindigkeitsprobleme auf, die sich nur über schnelle Assemblerroutinen lösen lassen. Doch wie übernehmen Sie Assemblerprogramme in ein Quick-Basic-Programm? Wie benutzen Sie MS-DOS- oder BIOS-Funktionen?

Quick Basic ähnelt stark Turbo Basic. Es gibt jedoch Besonderheiten, die gerade beim Einbinden von Maschinenspracheprogrammen auftreten.

Wie bei Turbo Basic, so ist auch bei Quick Basic eine gewisse Erfahrung mit Maschinenspracheprogrammen erforderlich. Als Beispiel wird hier, wie in dem Beitrag zu Turbo Basic, eine Routine verwendet, die einen bestimmten Bildschirmbereich nach links scrollt. Durch die Verwendung derselben Routine werden gleichzeitig einige Unterschiede zwischen Quick Basic und Turbo Basic aufgezeigt.

Um ein Assemblerprogramm in Quick Basic einbinden zu können, muß das Programm als Unterprogramm »public Name« und als »far« deklariert werden (Listing 1).

Wie Sie sicher bereits bemerkt haben, übergibt Quick Basic die Variablen bei einem Aufruf mit »call« nur als Offset (2 Byte). Der Segmentanteil der Variablen wird beim Aufruf der Routine von Quick Basic automatisch in das Datensegment DS geladen und mit übergeben. (Falls es für Sie notwendig ist, die Segmentadressen mit zu übergeben, benutzen Sie den Befehl »calls« und ändern das Assemblerprogramm dementsprechend um.) Wie der Stapel aussieht, zeigt das Bild.

Beim Rücksprung in das Hauptprogramm müssen Sie darauf achten, daß der Stapel frei von Adressen der Übergabeparameter ist. Das heißt: Bei der Übergabe von drei Variablen (3 mal der Offset = 6 Byte) muß der Befehl »ret« beim Rücksprung den Direktwert 6 (ret 6 = 6 Byte) enthalten.

Es ist empfehlenswert, bei einer Prozedur, die von einer Microsoft-Hochsprache aufgerufen wird, die Register SI, DI, SS, DS zu sichern, falls sie verwendet werden. Dies realisieren Sie am besten mit den Befehlen »push« und »pop«.

Demjenigen, dem die Funktion des Stapels nicht sehr geläufig ist, sei gesagt, daß der Stapel nach dem »Lifo-Prinzip« (»Last in, first out«) arbeitet, das heißt, der Wert, der zuletzt auf den Stapel kommt, wird als erster wieder vom Stapel gelesen.

Die Prozedur muß nach dem Fertigstellen noch mit einem Assembler (MASM oder TASM) zu einer Objektdatei assembliert werden. Ebenso muß das aufrufende Basic-Programm mit dem Compiler »bc.exe« außerhalb des Quick-Basic-Editors zu einer Objektdatei compiliert werden. Danach binden Sie beide Objektdateien mit dem Linker zusammen.

Um das Programm zu erzeugen, gehen Sie wie folgt vor:

Als erstes geben Sie das Basic-Programm ein und speichern es unter dem Namen »scrollb.bas« (Listing 2). Nachdem Sie den Editor verlassen haben, compilieren Sie es, indem Sie die Datei »bc« mit dem Parameter des Programmnamens aufrufen, also

```
bc scrollb
```

Anschließend binden Sie Ihr Programm und die Scroll-Routine folgendermaßen zusammen:

```
link scrollb+sleft,scrolle
```

Jetzt rufen Sie das fertige Exe-Programm mit dem Namen »scrolle« auf.

Es sollte noch angemerkt werden, daß Sie die Sleft-Routine noch verbessern können, indem Sie zum Beispiel die Variablen »Local« definieren (nähere Erläuterungen hierzu finden Sie im Quick-Basic-Handbuch).

Im Gegensatz zu einer Prozedur brauchen Sie zum Aufruf einer Funktion keinen Call-Befehl, Sie behandeln diese Funktion einfach wie einen vorhandenen Befehl, wie folgendes Beispiel zeigt:

```
print "10+5 ist";
print addiere (10,5)
```

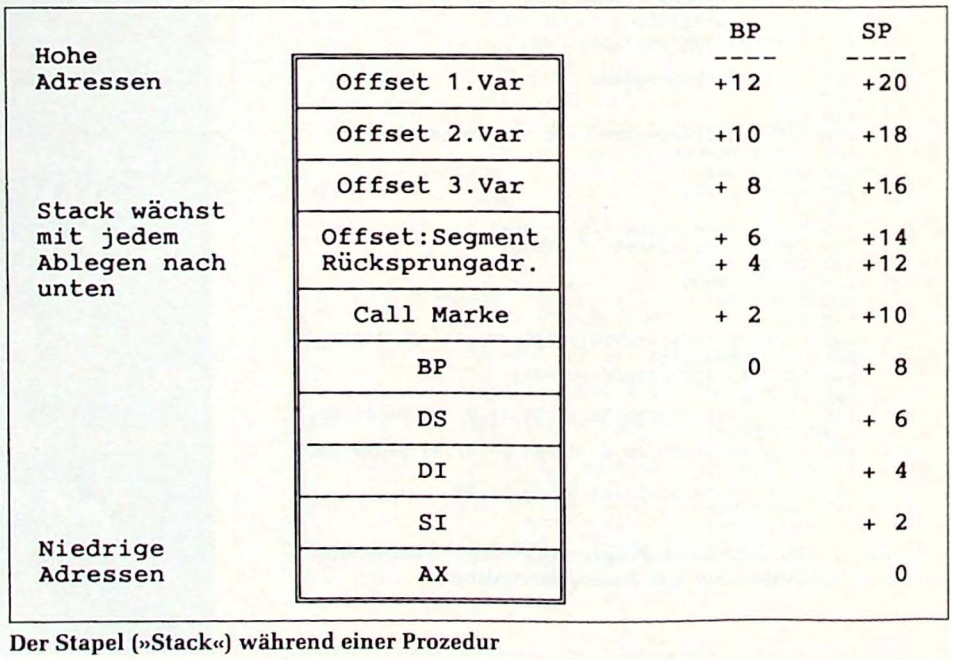
Bei einer Funktion werden Rückgabewerte aus der Maschinencoderoutine erwartet, die, wie in Tabelle 1 beschrieben, übergeben werden.

Listing 3 zeigt als Beispiel ein Assemblerlisting, das es erlaubt, Integer-Zahlen bis 65535 zu addieren. Die Parameterübergabe und das Verlassen erfolgt wie bei einer Prozedur. Im Quick-Basic-Programm müssen Sie die Funktion noch mit

```
declare funktion addiere (A%,B%)
```

deklarieren. Zum Einbinden der Funktion in das Programm sind dieselben Schritte notwendig wie bei einer Prozedur. Der Aufruf von MS-DOS- beziehungsweise BIOS-Funktionen ist in Quick Basic der gleiche. Durch die Verwendung von BIOS-Funktionen bietet sich Ihnen die Möglichkeit, in einem Programm das Vorhandensein einer oder mehrerer Festplatten, einer Grafikkarte oder eines Druckers festzustellen. Auch ist es möglich, spezielle Hardware-Eigenschaften eines peripheren Gerätes anzusprechen.

Als Beispiel soll eine BIOS-Routine, die einen definierten Bereich nach unten scrollt, verwendet werden. Hierbei wird die Interrupt-Nummer



Der Stapel (»Stack«) während einer Prozedur

10hex und die Funktionsnummer 7hex (Seite abwärts scrollen) benötigt. Eine Sammlung aller benötigten Parameter finden Sie in Tabelle 2.

Beim Start von Quick Basic ist es wichtig, die Bibliothek »qb.lib« mitzuladen, was Sie mit

```
qb Programmname /L
```

erreichen. Der Grund dafür besteht darin, daß die nötigen Routinen für den Aufruf von »call interrupt« darin enthalten sind. Listing 4 erläutert diesen Aufruf genauer.

Da von Quick Basic aus nicht direkt auf Prozessorregister zugegriffen werden kann, müssen dafür spezielle Variablen eingerichtet werden. Um einen Datentyp zu erhalten, der mehrere Elemente enthält, müssen Sie diesen mit »type xxxx« definieren (Zeilen 1 bis 12 in Listing 4). Registerwerte bleiben bis zur Zeit ihres Aufrufs als Variablen im Datensegment von Quick Basic gespeichert. Diese Variablen werden einem Datentyp zugeordnet, der beim Aufruf mit allen seinen Elementen übergeben wird.

In Zeile 13 wird ein Datentyp definiert und eine Elementliste zugeordnet. Hier wurden »inreg« und »outreg« als Datentypen definiert und bekommen die Elemente »regtyp« zugewiesen. »inreg« enthält die Registerwerte, die zur Ausführung der Funktion nötig sind, während Sie in »outreg« die Registerwerte nach Ausführung der Funktion (Rückgabewerte beziehungsweise Fehlercodes) finden.

Falls Sie in einer Funktion Segmentregister übergeben müssen, brauchen Sie dazu den Befehl »call interruptx«. Hier müssen Sie aber äußerste Vorsicht walten lassen! Der Befehl »call interruptx« sollte nur verwendet werden, wenn der Systemaufruf Segmentregisterwerte erfordert. Die falsche Änderung von Segmentregistern kann schwerwiegende, nicht wieder zu behebende Probleme verursachen.

Die Syntax des Befehls »call interrupt« sieht wie folgt aus:

```
Call Interrupt (Interruptnr,inreg,outreg)
Call Interruptx (Interruptnr,inreg,outreg)
```

»Interruptnr« steht für die Nummer des Interrupt (zulässiger Bereich ist hier von 0 bis 255).

Wenn Sie selbst ein bißchen weiter experimentieren und sich von (sicher vorkommenden) Systemabstürzen nicht abschrecken lassen, werden Sie feststellen, daß Sie durch die Möglichkeit, Maschinensprachepro-gramme in Quick Basic einzubinden, einerseits den Vorteil einer Hochsprache nutzen können und andererseits auf die Schnelligkeit der Assemblersprache nicht verzichten müssen.

(Guido Neuburg/Peter Seidl/wn)

Datentyp	Zurückgegebene Register
2-Byte Ganzzahl(%)	AX
4-Byte Ganzzahl(&)	Höherwertiger Teil in DX
	niederwertiger Teil in AX
Alle anderen Typen	Kurzer Offset in AX

Tabelle 1. In diesen Registern werden die Datentypen zurückgegeben

Parameter	Erklärung
AH	Funktionsnummer (7)
AL	Anzahl der Zeilen, die im definierten Bereich gescrollt werden
	AL = 0 Der ganze Bereich wird gelöscht
	Frei werdende Zeilen im Bereich werden gelöscht
CH	Zeile links oben
CL	Spalte links oben
DH	Zeile rechts unten
DL	Spalte rechts unten
BH	Attribut für gelöschte Zeilen

Tabelle 2. Die verschiedenen Parameter und ihre Bedeutung

Zusammenfassung für Quick-Basic-Aufruf-vereinbarungen:

a) Parameter werden auf dem Stapel in der gleichen Reihenfolge plaziert, in der sie im Basic-Quellcode erscheinen. Der erste Parameter liegt ganz oben im Speicher (weil er auch der erste auf dem Stapel abzulegende Parameter ist und der Stapel nach unten wächst).

b) Standardmäßig werden Basic-Parameter in Form von Zwei-Byte-Adressen als Referenz übergeben.

c) Beim Verlassen muß die Prozedur den Stapelzeiger (»Stackpointer«) SP auf den Wert zurücksetzen, den dieser vor der Plazierung der Parameter auf dem Stapel hatte. Dies wird von dem Befehl »ret n« ausgeführt, wobei n die Gesamtgröße aller Parameter in Byte ist.

d) Quick Basic beachtet bis zu 40 Zeichen eines Namens, während der Assembler nur die ersten 31 Zeichen beachtet (dies sollte allerdings kaum Probleme verursachen).

```
[W22]    1: ;scroll.asm
[2F6]    2: ;(c) Redaktionsbuero Everts & Hagedorn
[H22]    3: code segment
[EA2]    4: assume cs:code
[MZ1]    5: public sleft
[392]    6: sleft proc far
[7N1]    7: call marke
[ØØØ]    8:
[YU4]    9: ;Variablen der Scroll-Routine
[DX1]   1Ø:          anfang    dw Ø
[R32]   11:          spalten   db Ø
[9Y1]   12:          zeilen    db Ø
[ØØØ]   13:
[PH2]   14: marke proc near
[TB1]   15:      push bp
[7T1]   16: ;BP retten
[7S1]   17:      mov  bp,sp
[8A4]   18: ;BP mit Wert von SP laden
```

```
[BC1]   19:      push ds
[TT1]   2Ø: ;DS retten
[3B1]   21:      push di
[HS1]   22: ;DI retten
[GC1]   23:      push si
[YT1]   24: ;SI retten
[EN2]   25:      mov  di,[bp+12]
[GU5]   26: ;Offset des Wertes Ecke oben links
[H42]   27:      mov  ax,[di]
[TV5]   28: ;Wert (adressiert mit DS) in AX laden
[8N2]   29:      mov  di,[bp+1Ø]
[S86]   3Ø: ;Offset des Wertes Ecke unten rechts
[M42]   31:      mov  bx,[di]
[WV5]   32: ;Wert (adressiert mit DS) in BX laden
[9E2]   33:      mov  di,[bp+8]
[L46]   34: ;Offset des Wertes der Graphikkarte
```

Listing 1. Auf diese Weise wird das Assemblerlisting »scroll.asm« in Quick-Basic-Programme eingebunden

```
[R42]   35:       mov  cx,[di]
[ZV5]   36: ;Wert (adressiert mit DS) in CX laden
[XD2]   37:       mov  di,[bp+2]
[ØP4]   38: ;Offset der Variablen in DI
[MS1]   39:       mov  es,cx
[W24]   4Ø: ;ES Offset Graphikkarte
[HS1]   41:       mov  ds,cx
[T24]   42: ;DS Offset Graphikkarte
[PC1]   43:       push ax
[4U1]   44: ;AX retten
[NM1]   45:       sub  bl,al
[T51]   46:       dec  bl
[MP1]   47:       mov  bp,di
[UV2]   48:       mov  cs:[bp+2],bl
[FG5]   49: ;Spalten in Variablen schreiben
[TL1]   5Ø:       sub  bh,ah
[MV2]   51:       mov  cs:[bp+3],bh
[L35]   52: ;Zeilen in Variablen schreiben
[5T1]   53:       mov  bl,16Ø
[ME1]   54:       mov  cl,8
[ZP1]   55:       shr  ax,cl
[Z81]   56:       mul  bl
[LA1]   57:       pop  bx
[396]   58: ;den gepushten Wert von AX in BX holen
[VL1]   59:       sub  bh,bh
[KC1]   6Ø:       shl  bl,1
[TM1]   61:       add  ax,bx
[KS1]   62:       mov  si,ax
[5R1]   63:       mov  di,ax
[BB1]   64:       add  si,2
[FR2]   65:       mov  cs:[bp],di
[ULØ]   66:       cld
[BR1]   67:       sub  bx,bx
[FT2]   68:       mov  bl,cs:[bp+2]
[HG1]   69: schleife:
[U42]   7Ø:       mov  dx,[di]
[ØT1]   71:       mov  cx,bx
[RT1]   72:       rep  movsw
[F22]   73:       mov  [di],dx
[5X3]   74:       add  word ptr cs:[bp],16Ø
[KT2]   75:       mov  si,cs:[bp]
[WP1]   76:       mov  di,si
[BB1]   77:       add  si,2
[2V3]   78:       dec  byte ptr cs:[bp+3]
[E72]   79:       jnz  schleife
[AA1]   8Ø:       pop  si
[9H5]   81: ;SI den alten Wert zurückholen
[X81]   82:       pop  di
[WF5]   83: ;DI den alten Wert zurückholen
[5A1]   84:       pop  ds
[4H5]   85: ;DS den alten Wert zurückholen
[L91]   86:       pop  bp
[KG5]   87: ;BP den alten Wert zurückholen
[7C1]   88:       add  sp,2
[X75]   89: ;den CALL Marke vom Stack holen
[MN1]   9Ø: marke endp
[FUØ]   91:       ret 6
[MP7]   92: ;Far Return und Abräumen der Offset-Adresse
        n
[G23]   93: ;vom Stack (6 Byte).
[6P1]   94: sleft endp
[TF1]   95: code ends
[4MØ]   96: end
```

Listing 1. Auf diese Weise wird das Assemblerlisting »scroll.asm« in Quick-Basic-Programme eingebunden

```
[KØ2]    1: 'scrollb.bas
[UD6]    2: '(c) Redaktionsbuero Everts & Hagedorn
[YK5]    3: 'Da von Basic aus nicht direkt auf
[SV6]    4: 'Prozessor-Register zugegriffen werden
[3Ø7]    5: 'kann, müssen dafür spezielle Variablen
[7K3]    6: 'eingerichtet werden.
[TV5]    7: 'Um einen Datentyp zu erhalten, der
[CH6]    8: 'mehrere Elemente enthält, müssen Sie
[PT6]    9: 'diesen mit »type xxxx« definieren.
[W52]   1Ø: type regtyp
[Z12]   11:    ax as integer
[122]   12:    bx as integer
[322]   13:    cx as integer
[522]   14:    dx as integer
[912]   15:    bp as integer
[P12]   16:    si as integer
[SØ2]   17:    di as integer
[4M2]   18:    flags as integer
[P12]   19:    ds as integer
[S12]   2Ø:    es as integer
[WE1]   21: end type
[WN6]   22: 'Da Prozessor-Register nur 16 Bit groß
[725]   23: 'sind, werden alle Elemente als
[PY3]   24: '»integer« abgegeben.
```

```
[E96]   25: dim inreg as regtyp, outreg as regtyp
[PNØ]   26: cls
[252]   27: print "........."
[252]   28: print "........."
[196]   29: print ".........":'Ausgabe Bildschirmdaten
[252]   3Ø: print "........."
[252]   31: print "........."
[L52]   32: inreg.ax=&H7Ø1
[M52]   33: inreg.bx=&H7ØØ
[562]   34: inreg.cx=&H5Ø5
[CC2]   35: inreg.dx=&H154Ø
[1A1]   36: input a$
[GS5]   37: call interrupt(&H1Ø, inreg, outreg)
[4MØ]   38: end
```

Listing 2. »scrollb.bas« ist das Hauptprogramm in Quick Basic, aus dem die Assemblerroutine aufgerufen wird

```
[5A1]    1: ;add.asm
[2F6]    2: ;(c) Redaktionsbuero Everts & Hagedorn
[HR6]    3: ;diese Routine erlaubt Integerzahlen bis
[1F6]    4: ;zu einem Ergebnis von 65535 zu addieren
[L22]    5: code segnent
[EA2]    6: assume cs:code
[Z92]    7: public addiere
[SP2]    8: addiere proc far
[TB1]    9:         push bp
[7Ø6]   1Ø: ;Eingangssequenz - Alten BP sichern
[7S1]   11:         mov  bp,sp
[CG4]   12: ;Setze Stapel-Rahmenzeiger
[ZF2]   13:         mov  bx,[bp+8]
[492]   14: ;Lade Arg1 in AX
[762]   15:         mov  ax,[bx]
[VF2]   16:         mov  bx,[bp+6]
[F92]   17: ;Lade Arg2 in CX
[D62]   18:         mov  cx,[bx]
[VM1]   19:         add  ax,cx
[B93]   2Ø: ;CX auf AX addieren
[L91]   21:         pop  bp
[ND4]   22: ;Alten BP wiederherstellen
[BUØ]   23:         ret 4
[ED7]   24: ;Rücksprung und 4 Bytes wiederherstellen
[F12]   25: addiere endp
[TF1]   26: code ends
[4MØ]   27: end
```

Listing 3. »add.asm« erlaubt es, Integer-Zahlen bis zu einem Wert von 65535 zu addieren

```
[Y12]    1: 'interr.bas
[X96]    2: '(c) Redaktionsbüro Everts & Hagedorn
[W52]    3: TYPE regtyp
[Z12]    4:         ax AS INTEGER
[122]    5:         bx AS INTEGER
[322]    6:         cx AS INTEGER
[522]    7:         dx AS INTEGER
[912]    8:         bp AS INTEGER
[P12]    9:         si AS INTEGER
[SØ2]   1Ø:         di AS INTEGER
[4M2]   11:         flags AS INTEGER
[P12]   12:         ds AS INTEGER
[S12]   13:         es AS INTEGER
[WE1]   14: END TYPE
[E96]   15: DIM inreg AS regtyp, outreg AS regtyp
[PNØ]   16: CLS
[DU4]   17: 'Ausgabe von Bildschirmdaten
[CR2]   18: PRINT "............."
[CR2]   19: PRINT "............."
[CR2]   2Ø: PRINT "............."
[CR2]   21: PRINT "............."
[CR2]   22: PRINT "............."
[ØA2]   23: inreg.ax = &H7Ø1
[3A2]   24: inreg.bx = &H7ØØ
[6A2]   25: inreg.cx = &H5Ø5
[HD2]   26: inreg.dx = &H154Ø
[1A1]   27: INPUT a$
[GS5]   28: CALL interrupt(&H1Ø, inreg, outreg)
[4MØ]   29: END
```

Listing 4. »interr.bas« liest die Interrupt-Werte aus

Tuning für Turbo

Einbinden von Maschinenprogrammen in Turbo Basic

Obwohl Turbo Basic erheblich schneller ist als die meisten anderen Hochsprachen, können sich dennoch Problemstellungen ergeben, bei denen die Geschwindigkeit trotzdem nicht ausreicht (oder dem Programmierer durch Sprachdefinitionen Grenzen gesetzt sind). Es steht Ihnen jedoch der Weg offen, Maschinenprogramme und MS-DOS- beziehungsweise BIOS-Aufrufe direkt in Turbo Basic einzubinden.

Der Umgang mit Maschinensprache-programmen (jeder, der schon etwas Erfahrung mit der Programmierung in Assembler besitzt, wird dem sicher zustimmen) kann sich etwas kompliziert und kritisch gestalten. Ein falscher Befehl oder ein vertauschtes Byte – und der einzige Ausweg ist der altbekannte »Affengriff« ([Alt-Ctrl-Del]).

Eine Möglichkeit, um Maschinencoderoutinen in Turbo Basic einzubinden, ist die Inline-Funktion. Eine Maschinencoderoutine, die als Assembler-Inlinecode codiert ist, wird innerhalb des restlichen Programmcodes gespeichert und wie eine normale Prozedur aufgerufen.

Mit dem Compiler-Befehl »$inline« lassen sich – im Gegensatz zu den Befehlen »bload« oder »poke«, die ein Maschinenprogramm in einen eigenen Speicherbereich laden – Maschinensprachebefehle direkt in ein Programm einfügen.

Der Inline-Befehl darf nur in einer Prozedur verwendet werden, die als »inline« deklariert ist. In einer Inline-Prozedur können beliebig viele $inline-Compiler-Befehle enthalten sein.

Die Syntax des Inline-Befehls sieht folgendermaßen aus:

```
Bytefolge \ Dateinamen-Konstante
```

»Bytefolge« ist hier eine Folge von Integer-Konstanten, die durch Kommata getrennt sind. Der zulässige Wertebereich liegt zwischen 0 und 255. Normalerweise werden diese Konstanten in hexadezimaler Schreibweise angegeben (zum Beispiel »&H1A,&H2B.....«).

»Dateinamen-Konstante« ist der Name einer Datei, die mit einem Assembler erzeugt wurde; davon sind innerhalb eines Programms maximal 15 verschiedene zulässig. Diese Datei wird vom Compiler byteweise gelesen und ohne weitere Interpretation an der Stelle des Inline-Befehls in den Objektcode aufgenommen.

Aber Vorsicht: Folgende Register dürfen in einer Maschinencoderoutine, die als Assembler-Inlinecode geschrieben ist, nicht verändert werden oder müssen vor und nach einem Aufruf denselben Wert haben: Stacksegment/Stackpointer (SS/SP), Datensegment (DS) und das Register BP. Durch Indizierung über BP können eventuell übergebene Parameter vom Maschinenprogramm gelesen und/oder verändert werden.

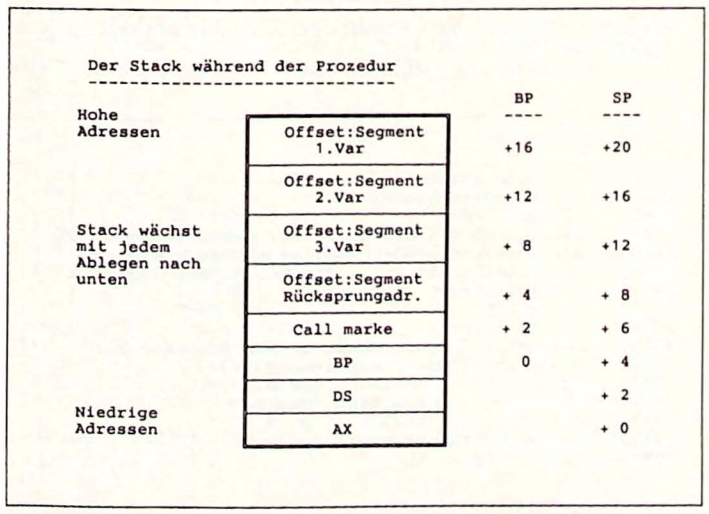


Bild 1. Der Stack-Aufbau während der Prozedur

Die Maschinenroutine darf nicht mit dem Return-Befehl beendet werden, da sich Turbo Basic den notwendigen Code zum Beenden einer Inline-Prozedur automatisch erzeugt. Wenn eine Inline-Prozedur vorzeitig verlassen werden soll, muß eine Verzweigung, die zum Ende des Maschinencodes führt, erzeugt werden.

In Listing 1 wird der Inhalt eines Fensters nach links gescrollt, wobei die links verschwindenden Zeichen rechts wieder erscheinen. Diese Prozedur im Hexdump ist in der Wirkung identisch mit dem noch folgenden Assemblerlisting (Listing 2).

Um die Adresse der Grafikkarte festzustellen, ist folgende Befehlszeile in Listing 1 erforderlich:

```
if (peek(&H410)and &H30) <> &H30 then segment%=&HB800 else segment%=&HB000
```

Diese Prozedur rufen Sie dann wie folgt auf:

```
call sleft (&H0000,&H184E,segment%)
```

»&H0000« bezeichnet die linke obere Ecke, »&H184E« die rechte untere Ecke und die Variable »segment%« enthält die Basisadresse Ihrer Grafikkarte.

Der Inline-Befehl erlaubt das Einbinden von Maschinensprache in Turbo-Basic-Programme

Die Parameter werden als Zeiger mit 32 Bit (Segment:Offset) in der Reihenfolge von links nach rechts auf den Stack gebracht (Bild 1).

Das ist aber noch nicht alles! Selbst komplette Maschinenprogramme des Typs »com« binden Sie mit Hilfe des Inline-Befehls in ein Turbo-Basic-Programm ein.

Als erstes erzeugen Sie eine Com-Datei mit Hilfe des MS-DOS-Programms »Debug«. Nach dem Start von »Debug« benutzen Sie »A« (für Assembler) und geben die Maschinenbefehle ein, danach »N« (für Name) und wählen einen Namen für die zukünftige Datei, der mit ».com« enden muß. Über »RCX« tragen Sie die Anzahl der zu schreibenden Bytes ein und speichern die Datei mit »W« (für Write) ab.

Selbstverständlich ist es auch mit Hilfe eines Makroassemblers möglich, lauffähige Com-Dateien zu erzeugen. Dazu benötigen Sie vier Programme: Einen Editor zur Eingabe des Quelltextes, den Makroassembler zum Übersetzen des Quelltextes in eine Objektdatei, einen Linker (das Programm »Link« auf der MS-DOS-Diskette), der eine oder mehrere Objektdateien in eine Exe-Datei zusammenbindet, und schließlich das MS-DOS-Programm »exe2bin«, um

die Exe-Datei in eine Com-Datei umzuwandeln.
Direkte Interrupt-Aufrufe innerhalb von Turbo Basic erlauben es, benötigte MS-DOS- beziehungsweise BIOS-Funktionen direkt anzusprechen. Die Syntax dieser Aufrufe sieht wie folgt aus:

```
call interrupt n
```

wobei »n« ein Integer-Ausdruck im Bereich von 0 bis 255 ist und die Nummer des Interrupts repräsentiert. Die Interrupts 10 (dezimal: 16) bis 1Fhex (dezimal: 31) sprechen die BIOS-Routinen des Systems direkt an. Der Interrupt 21hex (dezimal: 33) ist dabei der umfangreichste des Systems: Über ihn werden die Funktionen des MS-DOS-Betriebssystems angesprochen.

Auch der Aufruf von Interrupts bereitet in Turbo Basic keine Probleme

Da MS-DOS weit mehr als eine Funktion hat, muß eine zusätzliche Unterscheidung getroffen werden: Welche MS-DOS-Funktion ausgeführt werden soll, wird im oberen Byte des AX-Registers (AH) festgelegt. Um die Inhalte des Puffers für die Prozessorregister lesen beziehungsweise setzen zu können, gibt es den Befehl und die Funktion »reg«. Die Syntax dieses Befehls lautet wie folgt:

```
x% = reg(Register) (Funktion) reg Register, Wert (Befehl)
```

In »x%« stehen die Registerwerte beim Verlassen der Assemblerroutine. »Register« ist ein Integer-Ausdruck, sein Wert bezeichnet eines der Register des Prozessors, dessen Inhalt im Puffer »reg« gespeichert ist.
»Wert« ist ein Integer-Ausdruck im Bereich von 0 bis 65535 (16 Bit). Dieser Puffer enthält die Registerinhalte des Prozessors nach Ausführung der jeweils letzten (mit »call interrupt« aufgerufenen) Maschinenspracheroutine. Er kann als Array mit zehn Elementen vom Typ Integer aufgefaßt werden (Bild 2). Wenn der Wert von »Register« nicht im Bereich von 0 bis 9 liegt, erzeugt »reg« den Laufzeitfehler 5 (illegaler Funktionsaufruf).
»reg« als Befehl erwartet zwei Parameter und setzt ein Element des Puffers.

```
reg 1,&H0719
```

Der Befehl setzt das Element 1 des Puffers (AX) auf den Wert 719hex. Auch hier muß betont werden, daß dadurch kein Prozessorregister direkt geladen wird – »reg« setzt lediglich Anfangsbedingungen für den nächsten Aufruf mit »call« beziehungsweise liest das Ergebnis des letzten Aufrufs.
»reg« als Funktion erwartet als Argument die Nummer des Registers und liefert einen Integer-Wert zurück. Der Befehl

```
x% = reg(5)
```

setzt die Variable »x%« auf den zuletzt zurückgelieferten Wert des SI-Registers (nicht auf den tatsächlichen Wert von SI zum Zeitpunkt der Ausführung dieses Programmschritts). Listing 3 zeigt ein Turbo-Basic-Listing, das eine komplette Bildschirmseite nach unten scrollt. Dabei wird die Technik der Interrupt-Aufrufe deutlich gezeigt.

REG-Nummer	= Prozessor-Register
0	Flag-Register
1	AX
2	BX
3	CX
4	DX
5	SI
6	DI
7	BP
8	DS
9	ES

Bild 2. Die Bedeutung der verschiedenen Reg-Nummern im Register

Wie Sie sehen, eignet sich die Sprache Turbo Basic hervorragend zum Einbinden von Maschinenprogrammen. Mit wenigen Befehlen binden Sie Assemblerroutinen in Ihr Turbo-Basic-Programm ein. Daß Sie hierbei Assemblerkenntnisse mitbringen müssen, ist allerdings unerläßlich.

(Guido Neuburg/Peter Seidl/wn)

Einige Regeln zum Erzeugen von Com-Dateien für Turbo-Basic-Programme:

- Kein Stack-Segment im Programm definieren. Beim Linken wird zwar die Warnung »No Stack Segment« ausgegeben, die in diesem Fall jedoch ignoriert werden kann.
- Innerhalb des Programms dürfen Sprünge und Datenadressierungen nur relativ erfolgen.
- Die Prozessorregister DS (Datensegment), SS (Stapelsegment), SP (Stackpointer) und BP (Basepointer) müssen nach der Ausführung des Programms dieselben Werte wie beim Aufruf haben. Falls eines dieser Register nach der Programmausführung einen anderen Wert aufweist, besteht eine hohe Wahrscheinlichkeit, daß das System abstürzt.
- Lokale Daten sollten am besten auf dem Stack (Stapel) gehalten und über den jeweiligen Wert des SP adressiert werden.
- Wenn ein Programm als Com-Datei verwendet werden soll, muß dem ersten Befehl die Org-Anweisung »ORG 100H« vorausgehen.

```
[VS2]    1: sub sleft inline
[B67]    2: $inline &HE8,&H4,&HØ,&HØ,&HØ,&HØ,&HØ,&H55,&
            H8B,&HEC
[PH7]    3: $inline &H1E,&HC4,&H7E,&H1Ø,&H26,&H8B,&H5,&
            HC4,&H7E,&HC
[DN7]    4: $inline &H26,&H8B,&H1D,&HC4,&H7E,&H8,&H26,&
            H8B,&HD,&H8B
[5V7]    5: $inline &H7E,&H2,&H8E,&HC1,&H8E,&HD9,&H5Ø,&
            H2A,&HD8,&HFE
[4U7]    6: $inline &HCB,&H8B,&HEF,&H2E,&H88,&H5E,&H2,&
            H2A,&HFC,&H2E
[RK7]    7: $inline &H88,&H7E,&H3,&HB3,&HAØ,&HB1,&H8,&H
            D3,&HE8,&HF6
[ØV7]    8: $inline &HE3,&H5B,&H2A,&HFF,&HDØ,&HE3,&H3,&
            HC3,&H8B,&HFØ
[NM7]    9: $inline &H8B,&HF8,&H83,&HC6,&H2,&H2E,&H89,&
            H7E,&HØ,&HFC
[4T7]   1Ø: $inline &H2B,&HDB,&H2E,&H8A,&H5E,&H2,&H8B,&
            H15,&H8B,&HCB
[5F7]   11: $inline &HF3,&HA5,&H89,&H15,&H2E,&H81,&H46,
            &HØ,&HAØ,&HØ
[VL7]   12: $inline &H2E,&H8B,&H76,&HØ,&H8B,&HFE,&H83,&
            HC6,&H2,&H2E
[1K7]   13: $inline &HFE,&H4E,&H3,&H75,&HE3,&H1F,&H5D,&
            H83,&HC4,&H2
[HA1]   14: end sub
```

```
[4NA]   15: IF (PEEK(&H41Ø)AND &H3Ø) <> &H3Ø THEN segme
            nt%=&HB8ØØ ELSE segment%=&HBØØØ
```

Listing 1. »fensterb.bas« scrollt den Bildschirminhalt nach links

```
[H22]    1: code segment
[EA2]    2: assume cs:code
[A61]    3: org 1ØØh
[LE7]    4: ;weist den Assembler an, den Objektcode für
[2F6]    5: ;die ;folgenden Befehle ab der Speicher-
[SU6]    6: ;stelle 256 Bytes vom Beginn des Segments
[SR1]    7: ;abzulegen
[E21]    8: start:
[7N1]    9: call marke
[ZR6]   1Ø: ;Aufruf eines UPROS um die Adressen der
[ØM3]   11: ;Variablen zu erhalten
[DX1]   12:           anfang    dw Ø
[R32]   13:           spalten   db Ø
[9Y1]   14:           zeilen    db Ø
[PH2]   15: marke proc near
[TB1]   16:      push bp
```

```
[7T1]    17: ;BP retten
[7S1]    18:       mov  bp,sp
[BA4]    19: ;BP mit Wert von SP laden
[BC1]    2Ø:       push ds
[TT1]    21: ;DS retten
[EM2]    22:       les  di,[bp+16]
[B76]    23: ;Segmentadr.u.Offset des Wertes Ecke
[EZ1]    24: ;oben links
[BT2]    25:       mov  ax,es:[di]
[AT2]    26: ;Wert in AX laden
[2M2]    27:       les  di,[bp+12]
[B76]    28: ;Segmentadr.u.Offset des Wertes Ecke
[M52]    29: ;unten rechts
[ET2]    3Ø:       mov  bx,es:[di]
[DT2]    31: ;Wert in BX laden
[XC2]    32:       les  di,[bp+8]
[666]    33: ;Segmentadr.u.Offset des Wertes der
[U92]    34: ;Graphikkarte
[HT2]    35:       mov  cx,es:[di]
[GT2]    36: ;Wert in CX laden
[XD2]    37:       mov  di,[bp+2]
[ØP4]    38: ;Offset der Variablen in DI
[MS1]    39:       mov  es,cx
[W24]    4Ø: ;ES Offset Graphikkarte
[HS1]    41:       mov  ds,cx
[T24]    42: ;DS Offset Graphikkarte
[PC1]    43:       push ax
[BN3]    44: ;Inhalt von AX retten
[NM1]    45:       sub  bl,al
[T51]    46:       dec  bl
[MP1]    47:       mov  bp,di
[UV2]    48:       mov  cs:[bp+2],bl
[G75]    49: ;Spalten in Variable schreiben
[TL1]    5Ø:       sub  bh,ah
[MV2]    51:       mov  cs:[bp+3],bh
[UW4]    52: ;Zeilen in Variable schreiben
[5T1]    53:       mov  bl,16Ø
[ME1]    54:       mov  cl,8
[ZP1]    55:       shr  ax,cl
[Z81]    56:       mul  bl
[LA1]    57:       pop  bx
[396]    58: ;den gepushten Wert von AX in BX holen
[VL1]    59:       sub  bh,bh
[KC1]    6Ø:       shl  bl,1
[TM1]    61:       add  ax,bx
[KS1]    62:       mov  si,ax
[5R1]    63:       mov  di,ax
[BB1]    64:       add  si,2
[FR2]    65:       mov  cs:[bp],di
[ULØ]    66:       cld
[BR1]    67:       sub  bx,bx
[FT2]    68:       mov  bl,cs:[bp+2]
[HG1]    69: schleife:
[U42]    7Ø:       mov  dx,[di]
[ØT1]    71:       mov  cx,bx
[RT1]    72:       rep  movsw
[F22]    73:       mov  [di],dx
[5X3]    74:       add  word ptr cs:[bp],16Ø
[KT2]    75:       mov  si,cs:[bp]
[WP1]    76:       mov  di,si
[BB1]    77:       add  si,2
[2V3]    78:       dec  byte ptr cs:[bp+3]
[E72]    79:       jnz  schleife
[5A1]    8Ø:       pop  ds
[NV4]    81: ;DS alten Wert zurückholen
[L91]    82:       pop  bp
[3V4]    83: ;BP alten Wert zurückholen
[7C1]    84:       add  sp,2
[X75]    85: ;den Call Marke vom Stack holen
[MN1]    86: marke endp
[TF1]    87: code ends
[4K1]    88: end start
[4MØ]    89: end
```

Listing 2. »fenster.asm« wird in eine Objektdatei umgewandelt und kann in ein Turbo-Basic-Listing eingebunden werden

```
[RL1]     1:    REG 1,&HØ719
[KH5]     2: 'AH=7 Funktion abwärts scrollen
[YG2]     3: 'AL=19H 25 Zeilen
[CL1]     4:    REG 2,&HØ7ØØ
[4T7]     5: 'Atribut für gel. Zeilen (weiß auf schwarz)
[TK1]     6:    REG 3,&HØØØØ
[TL3]     7: 'CH=Ø Zeile links oben
[UU3]     8: 'CL=Ø Spalte links oben
[CM1]     9:    REG 4,&H1847
[T86]    1Ø: 'DH=18H Zeile rechts unten (18H dez. 24)
[RA6]    11: 'DL=4fH Spalte rechts unten (4fH dez. 79)
[E53]    12:    CALL INTERRUPT &H1Ø
[Z47]    13: 'Unterbrechung BIOS Bildschirmein-/ausgabe
```

Listing 3. »scroll.bas« schiebt eine komplette Bildschirmseite nach unten

HYPERKEY 4.0

Der Tastatur-Manager

Nutzen Sie die Vorteile Ihres PC!

Mit dem speicherresidenten Makroprozessor HYPERKEY können alle Tastaturtypen mit beliebigen, verkettbaren Zeichenfolgen belegt werden: Anwendungen, Kürzel, Programmschablonen, etc. erscheinen auf Tastendruck.
Komfortables Editieren, Abspielen, Speichern, Sortieren und Suchen der Belegungen erfolgt über eine ergonomische Benutzeroberfläche mit dreistufiger Auswahlliste und kontextsensitiver Hilfestellung.
Minimaler Speicherplatzbedarf wird durch optimiertes Programmlayout, automatische Expanded-Memory-Unterstützung, integrierte DeInstall- und Speicherreservierungsfunktion und den Mini-Makroprozessor TINYKEY/PLAYBACK gewährleistet.

Weiterhin neu in der Version 4.0:

- Jetzt auch Zusammenarbeit mit Programmen, die im PC-Grafikmodus arbeiten und zwischenzeitlich den Tastaturpuffer löschen
- Unterstützung von MF-2-Tastaturen und Einstellung aller Tastaturtypen von XT/AT in Deutsch/Englisch durch einen Treiber-Generator
- Unterscheidung der Shifttasten rechts/links und beliebige Kombinationsmöglichkeiten der Belegungen mit ALT und CONTROL
- Verbesserte Editiermöglichkeiten für Makros: Kopieren, Packen, Verschieben und Kontrolle der Speicher-Ressourcen
- Steuerung und Konfigurieren von HYPERKEY durch Aufrufparameter, zum Beispiel für das automatische Laden einer Belegung beim Start
- Makro-Abspielkontrolle aus externen Programmen mit frei wählbarer, ständig sichtbarer Statusanzeige auf dem Monitor
- Quit-and-Play-Funktion und Ausgabe der Makrobelegungen auf dem Drucker oder in eine Textdatei
- Konvertierungsprogramm für Tastaturbelegungen von Hyperkey Plus auf die neue Version und ausführliche Bedienungsanleitung in Deutsch
- Inklusive universell verwendbarem Maustreiber

für alle PC/XT/AT unter MS-DOS
mit minimal 256 KByte RAM, DOS ab Version 2.11 bis 3.3

Und das alles zum alten Preis!

Zudem bieten wir gegen Einsendung des Handbuchs von Hyperkey Plus ein super-günstiges Upgrade auf HYPERKEY 4.0 an.

HYPERKEY 4.0 **DM 99,–***

5 1/4"-Disk. Best-Nr. 229 3 1/2"-Disk. Best-Nr. 235

Upgrade von 3.0 **DM 29,–***

5 1/4"-Disk. Best-Nr. 2601 3 1/2"-Disk. Best-Nr. 2602

* Unabhängig von der Anzahl der bestellten Programme berechnen wir für das Inland 4,– DM bzw. für das Ausland 6,– DM Porto und Verpackung.

Bitte benutzen Sie die Bestellkarte.

DMV-Verlag · Postfach 250 · 3440 Eschwege

DMV Daten- und Medienverlag

Turbo Pascal mit Assembler
Die perfekte Ergänzung

Es gibt programmtechnische Aufgabenstellungen, die sich nur in Maschinensprache effizient lösen lassen, da nur diese einen wirklich uneingeschränkten Zugriff auf alle Komponenten des Computersystems bietet. Für Pascal-Programmierer ist deshalb von Interesse, wie sie ganze Programme oder kleine Routinen in Maschinensprache in ein Pascal-Programm einbinden.

Dem Problem des Einbindens von Maschinensprache in einen Pascal-Quelltext begegnet Turbo Pascal mit mehreren Lösungen: Ganze Funktionen oder Prozeduren lassen sich entweder komplett in Maschinensprache verfassen oder Sie schreiben nur bestimmte Teile einer Funktion, einer Prozedur oder des Hauptprogramms in Maschinensprache. Dabei wird der Maschinencode entweder direkt in den Quelltext eingefügt oder extern definiert.

Am besten lassen sich die unterschiedlichen Verfahren an einem Beispiel verdeutlichen. Nehmen Sie an, Sie möchten eine Tabulatorfunktion in Turbo Pascal einbinden, die – wie der Tab-Befehl in Basic – die Bildschirmausgabe ab einer bestimmten Spaltennummer beginnt. Die Funktion soll als Funktionsargument einen Wert entgegennehmen, der eine der Bildschirmspalten zwischen 1 und 80 auswählt. Die Funktion gibt dann einen String aus Leerzeichen zurück, der bei der Ausgabe den Cursor genau in der gewünschten Spalte der aktuellen Zeile positioniert.
Wenn Sie die Funktion auf diese Weise programmieren, ist sie innerhalb eines Write-Kommandos problemlos verwendbar. Ein typischer Aufruf der zu entwickelnden Tab-Funktion sieht dann wie folgt aus:

```
writeln('Die Summe beträgt',
tab(50),mark,' DM.');
```

Die Funktion soll also einen Byte-Parameter übernehmen und daraus einen String erzeugen, der als Resultat zurückgeliefert wird. Die Funktion läßt sich wie folgt definieren:

```
function tab(spalte: byte): string;
external;
```

Dabei unterscheidet sich die Funktionsdeklaration nur im Hinblick auf das angefügte Schlüsselwort »external« von der Deklaration einer Funktion, die selbst in Pascal geschrieben ist. Das Schlüsselwort zeigt dem Pascal-Compiler an, daß die Funktion »außerhalb« (also auf Diskette oder Festplatte) als Maschinenprogramm deklariert ist. Das Maschinenprogramm muß dabei als Objektdatei vorliegen, darf also nur assembliert, aber noch nicht gebunden (»gelinkt«) sein.

Beim Einbinden von Maschinenprogrammen sind bestimmte Konventionen einzuhalten

Natürlich hat sich das Maschinenprogramm (und damit der Programmierer) an einige Konventionen zu halten. So muß das Codesegment den Namen »code« oder »cseg« und das Datensegment den Namen »data« oder »dseg« tragen; darüber hinaus dürfen Sie keine Segmente definieren. Es ist unzulässig, Variablen im Datensegment zu initialisieren. Das heißt, daß Sie bei der Reservierung von Werten nur das Fragezeichen verwenden dürfen. Alle Variablen, Prozeduren oder Funktionen, die von Pascal aus verfügbar sein sollen, müssen als »public« deklariert sein, die Prozedur oder Funktion in Assembler als Near-Prozedur, da sie von Turbo Pascal aus mit einem Near-Call aufgerufen wird.
Das sind die wichtigsten Regeln beim Schreiben externer Prozeduren oder Funktionen. Daraus können Sie nun bereits das Gerüst für die Tab-Funktion in Assembler entsprechend Bild 1 aufbauen.
Das Wichtigste fehlt aber bislang noch: Das eigentliche Programm. In das Programmgerüst ist daher noch der Assemblerquelltext für die Tabulatorfunktion einzubinden. Dabei taucht schnell das erste Problem auf: Wie kommt man an das Funktionsargument der Tab-Funktion?
Dazu muß man das Vorgehen von Turbo Pascal beim Aufruf von Funktionen oder Prozeduren kennen. Die Funktionsargumente werden dabei der Reihe nach (also von links nach rechts in der Deklaration) auf den Stapel (Stack) »geschoben«. Konstantenparameter (»call by value«) werden direkt übergeben während bei Variablenparametern (»call by reference«) ein Zeiger auf den Speicherplatz der Variablen übergeben wird. Konstantenparameter belegen daher je nach Datentyp unterschiedlich viel Platz auf dem Stack. Byte-, Wort-, Integer-, Boolean- und Char-Konstanten werden als Wörter übergeben. Bei den 8-Bit-Datentypen »Byte«, »Boolean« und »Char« liegt der Grund dafür in der Tatsache, daß die Prozessoren der 80x86-Reihe von Intel nur Wörter, nicht aber Bytes auf dem Stapel ablegen können. Die Byte-Information findet sich dann immer im unteren Byte des auf dem Stapel befindlichen Wortes, die Wahrheitswerte »true« und »false« sind dabei als 1 und 0 codiert. Strings, Mengen, Arrays, Records und Zeiger werden als Doppelwortzeiger auf dem Stack übergeben, zuerst der Offset und dann das Segment. Tabelle 1 faßt die Übergabeformen von Funktionsargumenten zusammen.
Funktionsergebnisse werden in den Prozessorregistern zurückgeliefert, wenn es sich um Integer-, Char-, Boolean- oder Real-Werte handelt. Alle Mengen, Arrays, Records oder Zeiger werden als Zeiger im Registerpaar DX:AX zurückgegeben. Tabelle 2 zeigt eine Zusammenfassung der verwendeten Register für die einzelnen Datentypen des Funktionsergebnisses.
Eine besondere Rolle spielen Strings (Zeichenketten). Soll eine Funktion als Ergebnis einen String liefern, so wird vor dem Funktionsaufruf ein Datenbereich für die Speicherung des Strings angelegt. Als erstes Element (vor allen Funktionsargumenten) gelangt dann ein Zeiger auf diesen Datenbereich auf den Stapel.
Im gegebenen Fall soll das Funktionsergebnis ein String sein. Daher wird beim Aufruf der Tab-Funktion zunächst ein Zeiger auf den Datenbereich des Strings auf dem Stapel plaziert. Danach legen Sie die einzelnen

```
code segment byte public ;Name des Codesegments
     assume cs:code      ;CS zeigt auf CODE
     public tab          ;Tab-Funktion wird deklariert
tab  proc near           ;Tab-Funktion muß NEAR-Prozedur sein
;
;hier Programm
;
tab  endp                ;Ende der Prozedur
code ends                ;Ende des Segments
     end                 ;Ende des Listings
```

Bild 1. Das Assemblergerüst für eine Tabulatorfunktion

Funktionsargumente auf dem Stapel ab. Da Sie der Tab-Funktion ein Byte als Konstantenparameter übergeben, wird ein Wort auf den Stack geschoben. Das niederwertige Byte dieses Wortes enthält dann den Wert des Funktionsarguments. Nach der Übergabe des Funktionsarguments rufen Sie die Funktion auf.

Um nun das Funktionsargument zu ermitteln, muß die Routine den entsprechenden Wert vom Stapel lesen. Unter Turbo Pascal darf eine Funktion alle Register verändern, nur die Werte für BP, SS, DS und SP müssen erhalten bleiben. Daher müssen Sie zunächst das BP-Register auf den Stapel retten. Mit BP als Zeiger lassen sich dann die Funktionsargumente vom Stapel lesen. Übertragen Sie dazu SP nach BP.

```
;BP retten
push bp
;BP zeigt auf Stapel
mov bp,sp
```

Addieren Sie anschließend den Wert vier zu BP, weil Turbo Pascal ein Wort (nämlich die Rücksprungadresse zum Pascal-Programm) zusätzlich auf den Stack schiebt und das eigene Programm ein weiteres Wort (also zwei Byte) für das Retten von BP belegt hat. BP zeigt nun auf das erste Funktionsargument auf dem Stapel.

Wenn Sie die einzelnen Funktionsargumente in die Register laden möchten, sollten Sie beachten, daß der Stack eine Lifo-Struktur (»last in, first out«) darstellt. Das heißt: Auch wenn die Funktionsargumente im Hinblick auf die Prozedurdeklaration von links nach rechts auf den Stapel gelegt wurden, muß sie die Maschinenroutine in der umgekehrten Reihenfolge wieder vom Stapel lesen.

Bei der Tab-Funktion spielt das keine Rolle; sie liefert nur ein Funktionsargument, nämlich »spalte«. Das ist nach der obigen Assemblersequenz an der Position SS:[BP + 4] verfügbar. An der Position SS:[BP + 6] befindet sich dann der Zeiger auf den Datenbereich des Strings, der als Resultat der Funktion gespeichert werden soll. Daher ist es ein leichtes, das Funktionsargument zu lesen und das Resultat der Funktion zu speichern. Um das Ganze ein wenig zu vereinfachen, können Sie sich mit

```
spalte equ byte ptr ss:[bp+4]
resultat equ dword ptr ss:[bp+6]
```

die symbolischen Namen »spalte« und »resultat« definieren, um mit ihnen das Funktionsargument und das Funktionsergebnis zu adressieren.

Was fehlt, ist der eigentliche Rumpf der Tab-Funktion. Aber auch das ist keine schwierige Aufgabe. Die Tab-Funktion soll einen String liefern, der so viele Leerzeichen enthält, daß die Ausgabe des Strings den Cursor an die Bildschirmspalte »spalte« bewegt. Sie lesen daher einfach die aktuelle Bildschirmspalte aus und ziehen diesen Wert von »spalte« ab. Dadurch erhalten Sie die Anzahl der Leerzeichen, die in den String einzufügen sind. Mit der Assemblersequenz gemäß Bild 2 ist dies schnell getan.

```
mov   ah,15          ;BIOS-Funktion 15
int   10h            ;Videostatus lesen
mov   ah,3           ;BIOS-Funktion 3
int   10h            ;Cursorposition lesen
mov   al,spalte      ;lade SPALTE nach AL
sub   al,dl          ;bilde Differenz
dec   al             ;ziehe eins ab
mov   cl,al          ;übertrage sie nach CL
xor   ch,ch          ;und lösche CH
les   di,resultat    ;lade Adresse des Strings
                     ;nach ES:DI
mov   al,cl          ;lade Stringlänge nach AL
cld                  ;String vorwärts speichern
stosb                ;speichere Stringlänge
mov   al,' '         ;Leerzeichen für den String
rep   stosb          ;speichere die erforderliche
                     ;Anzahl Leerzeichen
```

Bild 2. Der Assemblerrumpf der Tab-Funktion

Nach getaner Arbeit muß der Stapelspeicher aufgeräumt werden

Nun hat die Tab-Funktion einen entsprechenden String generiert. Dabei wurde der vom aufrufenden Programm eingerichtete Zeiger auf den String-Datenbereich verwendet. Als erstes Zeichen eines Strings speichert Turbo Pascal die Stringlänge, erst dahinter folgen die einzelnen Zeichen des Strings. Daher wurde zunächst mit dem Stosb-Befehl die Stringlänge gespeichert und später in der Rep-Schleife die erforderliche Anzahl an Leerzeichen hinzugefügt.

Zum Schluß müssen Sie noch dafür sorgen, daß der Stapelspeicher wieder aufgeräumt wird. Dazu sind alle Funktionsargumente vom Stack zu entfernen. Zuvor müssen Sie aber das zu Beginn der Funktion gesicherte BP-Register noch vom Stack holen. Danach wird die Funktion mit dem Ret-Kommando verlassen, das zusätzlich zwei Byte vom Stapel holt; die Bytes nämlich, die das Funktionsargument »spalte« belegt hat. Die Abschlußsequenz der Tab-Funktion sieht daher wie folgt aus:

```
;hole BP zurück
pop bp
;und räume Stapel auf
ret 2
```

Wenn Sie jetzt die einzelnen Sequenzen zusammenführen und sie in das vorgestellte Gerüst für die Tab-Funktion einsetzen, so ist die Funktion bereits fertig. Listing 1 zeigt den so erhaltenen Assemblerquelltext.

Um daraus eine Funktion zu erzeugen, die Turbo Pascal bei der Compilierung einbindet, übersetzen Sie den Quelltext mit einem Assembler, der Obj-Formate erzeugen kann. Dazu sind zum Beispiel der Microsoft-Assembler MASM oder der Turbo-Assembler TASM in der Lage. Nachdem Sie die Funktion als Datei »tab.obj« auf der Diskette oder Festplatte gespeichert haben, genügt folgende Anweisung in Ihrem Pascal-Programm:

```
{$l tab}
function tab(spalte: byte): string;
external;
```

Damit wird die Datei als Funktion »tab« in den Objektcode des Pascal-Programms eingebunden. Die Compiler-Direktive »$l« sorgt dafür, daß die nachstehende Obj-Datei in den Arbeitsspeicher gelesen wird. Dadurch ist die Tab-Funktion in Ihrem Programm einsetzbar. Listing 2 zeigt einen entsprechenden Quelltext.

Mit diesen Ausführungen sollten Sie nun in der Lage sein, selbst Maschinenprogramme in Ihre Pascal-Programme einzubinden. Dabei dürfen die Anzahl und die Datentypen der Parameter sowie die Funktionsergebnisse variieren.

Auf einfache Weise lassen sich solche Maschinenprogramme auch in eine Unit einbinden. Der wichtigste Unterschied besteht darin, daß die Prozedur dann nicht als Near-, sondern als Far-Programm definiert werden muß, da Funktionen und Prozeduren aus Units von Turbo Pascal mit einem »Far Call« aufgerufen werden. Die kurze Sequenz

```
unit Tabulator;
interface
function tab(spalte: byte):string;
implementation
{$l tab}
function tab; external;
end.
```

genügt bereits, um die Tab-Funktion als einzige Funktion einer Unit namens »Tabulator« zu deklarieren. Voraussetzung dafür ist natürlich, daß im Assemblerquelltext von »tab.asm« das Near-Kommando

durch ein Far-Kommando ersetzt wird.
Zugegebenermaßen war das Beispiel einer Tab-Funktion ein wenig »gekünstelt«. Erstens läßt sich eine solche Funktion leicht auch in Pascal formulieren, ohne nennenswerte Geschwindigkeitsverluste hinnehmen zu müssen, und zweitens lohnt sich die Verwendung von Funktionen oder Prozeduren im Obj-Format erst bei umfangreichen Programmen.
Für kleine, kurze Maschinenprogramme bietet Turbo Pascal eine wesentlich benutzerfreundlichere und einfachere Art, sie in ein Pascal-Programm einzubinden. Turbo Pascal erlaubt es nämlich, Objektcode direkt in einen Pascal-Quelltext zu schreiben.
Dazu wird die Inline-Anweisung benutzt. Mit dem Inline-Befehl lassen sich Maschinenbefehle direkt in ein Turbo-Pascal-Programm einbinden. Es werden dabei allerdings keine symbolischen Assemblerbefehle, sondern nur »echte« Maschinenbefehle, also Zahlen beziehungsweise Bitmuster, verarbeitet. Die einzelnen Zahlen trennen Sie durch einen Schrägstrich voneinander. So bindet die Zeile

```
inline($8b/$ec);
```

beispielsweise den Maschinencode für den Befehl »mov bp,sp« in ein Pascal-Programm ein. Der Compiler übernimmt während der Compilierungszeit den Maschinencode des Inline-Befehls direkt in den Objektcode des Pascal-Programms. Das Schöne am Inline-Kommando ist nun, daß auch Variablen in den Inline-Code mit aufgenommen werden dürfen. Verwenden Sie zum Beispiel die Sequenz

```
inline(status);
```

so wird an der entsprechenden Stelle im Programmcode die Adresse der Variablen oder Konstanten mit Namen »status« eingefügt. Alle globalen Variablen oder Konstanten stehen an der entsprechenden Offsetadresse im Datensegment. Lokale Variablen sind im Stacksegment gespeichert. Die Funktionsargumente einer Inline-Funktion oder Inline-Prozedur werden dabei als Relativwert zum BP-Register im Stapelspeicher abgelegt. Deklarieren Sie eine Prozedur

```
procedure demoproz (var anzahl: word;
adresse: ^word);
begin
...
end;
```

so können Sie auf die Variablen »anzahl« und »adresse« innerhalb des Inline-Rumpfes leicht zugreifen. Indem Sie nämlich die Namen der Variablen in den Inline-Code einsetzen, erzeugt Turbo Pascal für jede Variable den Offset relativ zum BP-Register. Wenn Sie also im obigen Beispiel die Inline-Sequenz

```
inline ($8b/$86/anzahl/
(* MOV AX,anzahl[BP] *)
$c4/$be/adresse);
(* LES DI,adresse[BP] *)
```

verwenden, wird an den Stellen, an denen die Variablennamen »anzahl« und »adresse« im Text vorkommen, automatisch ein Offset zum BP-Register berechnet. Dadurch kann eine Inline-Prozedur tatsächlich auf die Variablen und ihre Werte zugreifen.

Operatoren erleichtern dem Programmierer das Arbeiten mit Inline-Code

Es existieren zwei Operatoren, die sich im Zusammenhang mit Inline-Code gut einsetzen lassen. Es sind die Operatoren »<« und »>«. Mit ihnen wählen Sie den höher- oder niederwertigen Teil eines Wortes aus. Das kann sehr hilfreich sein. Nehmen Sie einmal an, Sie wollten in einer Inline-Prozedur die Zwei-Byte-Zahl 1000 auf das AH- und das CH-Register verteilen. Das AH-Register soll den höherwertigen Teil und das CH-Register den niederwertigen Teil der Zahl erhalten. Um nun für diese Ladeoperationen die Mov-Befehle mit AH beziehungsweise CH als Argumenten zu verwenden, muß dahinter – als Parameter – das Byte stehen, das nach AH oder CH geladen werden soll. Sie müssen also die Zahl 1000 in ein niederwertiges und ein höherwertiges Byte unterteilen. Dazu können Sie Ihren Kopf, ein Blatt Papier und einen Stift oder einen Taschenrechner benutzen. Mit dem Kleiner- und dem Größer-Operator hingegen haben Sie sofort »<1000« beziehungsweise »>1000« in den Inline-Befehl eingefügt, um das nieder- oder höherwertige Byte der Zahl 1000 auszuwählen.
Bei der Übergabe von Argumenten an eine Prozedur oder Funktion, die vollständig als Inline-Routine aufgebaut ist, verfährt Turbo Pascal prinzipiell wie beim Aufruf einer anderen Funktion oder Prozedur. Die einzelnen Schritte dazu wurden bereits weiter oben bei der Erläuterung des Aufrufs von externen Routinen vorgestellt.

Trifft der Turbo-Pascal-Compiler auf Inline-Code, so hat er prinzipiell zwei Möglichkeiten, die entsprechenden Maschinenbefehle in den Programmcode einzubinden. Einerseits können sie – wie bei einem Assemblermakro – direkt an der aktuellen Stelle in den Code eingefügt werden. Andererseits lassen sie sich – wie bei einem Unterprogrammaufruf – an einer Stelle im Programmcode einmal plazieren und von anderen Adressen aus mit einem »call« aufrufen. Welches Verfahren der Compiler anwendet, hängt von der Position des Inline-Befehls im Programm ab.

Steht das Inline-Kommando irgendwo inmitten einer Prozedur, einer Funktion oder des Hauptprogramms, wird der Code in jedem Fall an der entsprechenden Stelle direkt eingefügt. Das kann natürlich unter Umständen dennoch in einem (Pascal-) Unterprogramm sein, nur wird dann für den Inline-Code kein eigenes Unterprogramm erzeugt.

Erzeugen Sie hingegen eine Funktion oder Prozedur, die ausschließlich aus einer Inline-Anweisung besteht, generiert der Compiler auch ein Unterprogramm. Er verfährt immer dann auf diese Weise, wenn der Inline-Code durch eine Begin- und eine End-Direktive eingeschlossen ist. Möchten Sie beispielsweise durch Einbinden des Cld-Kommandos des Prozessors innerhalb eines Pascal-Programms bewirken, daß das Richtung-Statusbit gesetzt wird, so geschieht dies durch die folgende Inline-Anweisung:

```
inline($fc);
```

Plazieren Sie nun diese Zeile in eine eigene Prozedur ohne »Begin« und »End«, so erhalten Sie

```
procedure setdf; inline($fc);
```

Datentyp	auf den Stack übergeben als
byte	Wort (unteres Byte enthält Byte-Argument)
word	Wort (vorzeichenlos)
integer	Wort (mit Vorzeichen)
char	Wort (unteres Byte enthält Zeichencode)
boolean	Wort (unteres Byte enthält 0 oder 1)
longinteger	Doppelwort (Reihenfolge: niederwertiges, höherwertiges Wort)
real	Dreifachwort
pointer	Doppelwort (Reihenfolge: Offset, Segment)
array	Doppelwortzeiger auf den String-Datenbereich
record	Doppelwortzeiger auf den Record-Datenbereich
set	Doppelwortzeiger auf den Set-Datenbereich

Tabelle 1. Die Formate bei der Übergabe von Daten an eine Turbo-Pascal-Funktion

Jetzt auch im Abo

DOS EXTRA 7:

POWER FÜR PROGRAMMIERER

EXTRA Programmierung der Hercules- und CGA-Grafikkarten

EXTRA PC-Bausteine im Detail

EXTRA C-Grundlagen für Einsteiger und Profis

EXTRA Benutzeroberfläche und Präprozessor für SBasic

EXTRA Sprite-Editor zum Abtippen

EXTRA Komprimierte Programme mit dem DOS-Packer

...und vieles mehr

Best.-Nr. 384 **18,– DM**

DOS EXTRA 1 UND 2 SIND AUSVERKAUFT
NOCH ÜBER DEN VERLAG ERHÄLTLICH:

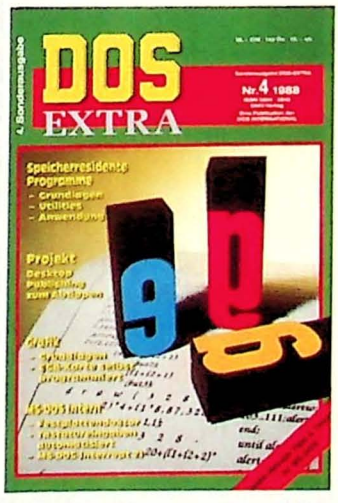
DOS EXTRA 3:
Für glasklare Informationen

- Der große DOS-Kurs »Der PC – glasklar in überarbeiteter Form
- Datenfernübertragungs-Programm in 100% Assemblercode
- Sprachdigitalisierung mit dem PC
- Hercules-Grafik-Toolbox in Turbo Pascal
- Grafikfunktionen für Turbo C

Best.-Nr. 356 **18,– DM***

DOS EXTRA 4:
Was Sie schon immer interessiert hat

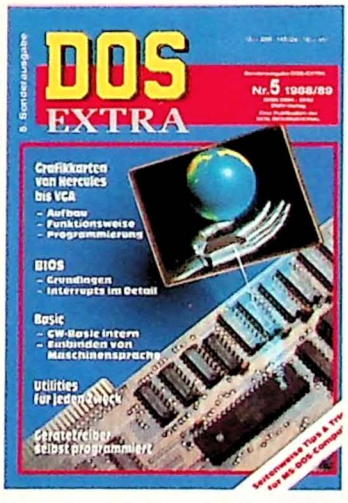
- Desktop Publishing selbstgemacht mit SDTP
- Speicherresidente Programme schreiben, analysieren und entfernen
- Alles über MS-DOS-Interrupt 21 hex
- Grundlagen zu Grafikkarten



Best.-Nr. 381 **18,– DM***

DOS EXTRA 5:
Von Basic bis BIOS

- Aufbau und Programmierung der wichtigsten Grafikkarten
- Basic für Einsteiger und Profis
- Die Interrupts des BIOS im Detail
- Gerätetreiber-Grundlagen



Best.-Nr. 382 **18,– DM***

DOS EXTRA 6:
Alles über Basic

- Schneller Basic-Compiler zum Abtippen
- GW-Basic selbst erweitert
- Basic intern: Token, Variablen, Speicherverwaltung
- Datenbankprogramm mit Menüsteuerung
- Turbo-Basic-Funktions-Bibliothek für die CGA-Karte
- Strukturierte Programmierung in Quick Basic 4.0

Best.-Nr. 383 **18,– DM**

Die Ausgaben DOS EXTRA 3, 4, 5 und 6 sind nur noch über den Verlag zu beziehen. Zu allen Ausgaben (DOS EXTRA 1-7) ist die Databox im Format 5 1/4" oder 3 1/2" MS-DOS erhältlich. Bitte benutzen Sie die Bestellkarte.

* Unabhängig von der Anzahl der bestellten Zeitschriften berechnen wir für das Inland 4,– DM bzw. für das Ausland 6,– DM Porto und Verpackung.

DMV-Verlag · Postfach 250 · 3440 Eschwege

DMV Daten- und Medienverlag

Toolboxen für Programmierer gesucht

Haben Sie eine Toolbox in C, Pascal, Basic, dBase, Clipper oder Assembler geschrieben? Beispielsweise schnelle, universelle Grafikroutinen, eine Textverarbeitungs- oder eine Datenbank-Toolbox? Dann sollten Sie sich schnellstmöglich mit uns in Verbindung setzen. Schicken Sie uns unverbindlich eine Diskette mit Ihrem Programm und einen Ausdruck der Dokumentation.

Falls Sie sich vorweg informieren möchten: Zwischen 10.00 und 12.00 Uhr und zwischen 15.00 und 17.00 Uhr stehen wir Ihnen an allen Werktagen unter der Rufnummer (089) 4 61 00 10 zur Verfügung.

Schicken Sie Ihre Unterlagen an folgende Adresse:

Redaktionsbüro
Everts & Hagedorn
»DOS Software«
Hans-Stießberger-Str. 3
8013 Haar/München

In diesem Fall wird der Inline-Code (das Byte FChex) an der Position direkt in das Compilat eingebunden, an welcher der Aufruf der Setdf-Prozedur im Programm auftritt. Benutzen Sie dagegen für die gleiche Prozedur die Deklaration

```
procedure setdf;
begin
inline($fc);
end;
```

so wird an der Stelle, an der ein Aufruf der Setdf-Prozedur erfolgt, ein Unterprogrammaufruf ausgelöst.
Eine völlig andere Form der maschinennahen Programmierung unter Turbo Pascal liegt in der Möglichkeit, Interrupts und Funktionen des Betriebssystems aufzurufen. Damit ist der Programmierer in der Lage, fast alle Fähigkeiten des Betriebssystems auch unter Pascal zu nutzen. Turbo Pascal bietet dazu den Msdos- und den Intr-Befehl an. Mit »Msdos« lassen sich Funktionen des Interrupt 21hex aufrufen, mit »Intr« wird jeder beliebige Interrupt aufgerufen. Die Syntax der Befehle lautet:

```
msdos(<Register-Variable>)
intr(<Interrupt-Nummer>,<Register-Variable>);
```

Beiden Anweisungen müssen Sie eine Variable übergeben, mit der die einzelnen Prozessorregister geladen werden. In Turbo Pascal ab Version 4.0 ist sie in der Unit »Dos« als Datentyp »Registers« vordefiniert, für ältere Pascal-Versionen müssen Sie sie selbst definieren:

```
type registers = record
case boolean of
true:
(ax,bx,cx,dx,bp,si,di,ds,es,flags:integer);
false: (al,ah,bl,bh,cl,ch,dl,dh: byte);
end;
```

Datentyp	zurückgegeben als
byte	Byte im AL-Register
word	Wort im AX-Register
integer	Wort im AX-Register
char	Zeichencode im AL-Register
boolean	0 oder 1 im AL-Register
longinteger	Doppelwort in DX:AX
real	Dreifachwort in DX:BX:AX1
pointer	Doppelwort in DX:AX
array	Doppelwortzeiger in DX:AX
record	Doppelwortzeiger in DX:AX
set	Doppelwortzeiger in DX:AX

Tabelle 2. Die Übergabe der Funktionsresultate an ein Turbo-Pascal-Programm

Wie Sie an der Deklaration erkennen, handelt es sich beim Registers-Datentyp also um einen Record. Sie sind damit in der Lage, alle Elemente des Datentyps – also die Register AX bis ES, die Register AL bis DH und das Kennzeichenregister »flags« – als Elemente einer Registervariablen anzusprechen. Um eine Funktion des Betriebssystems aufzurufen, definieren Sie vorher eine entsprechende Variable für den Zugriff auf die Prozessorregister, zum Beispiel:

```
var
cpu: registers;
```

Nach dieser Deklaration können Sie jedes Register des Prozessors einzeln ansprechen. Wenn Sie das Register AX benutzen möchten, verwenden Sie die Variable »cpu.ax«, für Register CL verwenden Sie »cpu.cl« und so weiter.

Über diese Prozessorvariable lassen sich nun Zuweisungen in beide Richtungen vornehmen. Sie übergeben damit den Funktionen des Betriebssystems vor dem Aufruf Parameter in den Registern oder erhalten nach dem Aufruf Ergebnisse in den Registern zurück. Häufig sind den Funktionen des Betriebssystems Datenpuffer oder ähnliches zu übergeben. Meist wird dazu die Segment- und die Offsetadresse des Speicherbereichs verlangt. Diese Datenpuffer lassen sich als String-Variablen definieren und die Adressen mit den Pascal-Kommandos »Ofs« und »Seg« ermitteln. Möchten Sie beispielsweise den Text »Ein schöner Tag mit Turbo Pascal« über die MD-DOS-Funktion 21hex ausgeben, kann das wie folgt geschehen:

```
program textaus;
uses dos;
var
text: string[100];
cpu: registers;
begin
text:='Ein schöner Tag mit Turbo Pascal'+#0;
cpu.ah:=9;
cpu.ds:=seg(text[1]);
cpu.dx:=ofs(text[1]);
msdos(cpu);
end.
```

Analog dazu arbeitet die Intr-Funktion, jedoch geben Sie hier mit dem Parameter »<Interrupt-Nummer>« noch die Nummer eines MS-DOS- oder BIOS-Interrupts an.

Leistungsfähige und schnelle Pascal-Programme lassen sich vor allem durch die Kombination von »echter« Maschinensprache und maschinennaher Pascal-Programmierung erreichen. Damit kann auch ein Pascal-Programmierer in Bereiche vorstoßen, die bisher noch den Assemblerprofis vorbehalten blieben.

(Martin Althaus/zi)

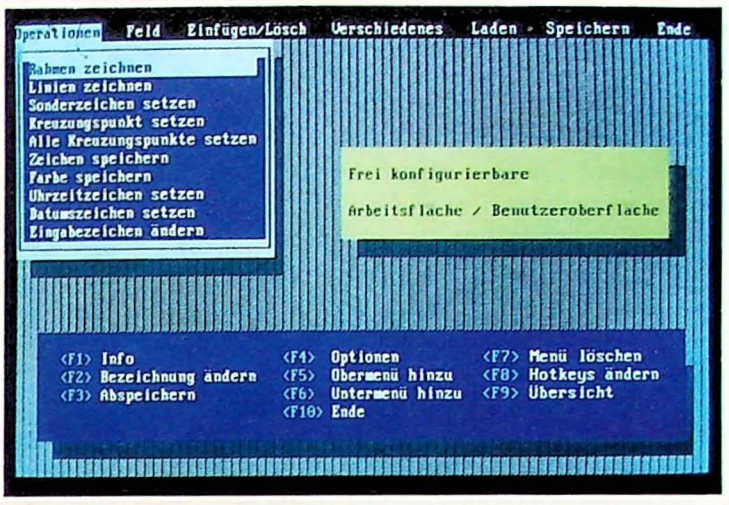
```
[ZØ4]    1: ; Programmname: TAB.ASM
[WY3]    2: ; (C) 1989 Everts & Hagedorn
[8U3]    3: ;           DOS International
[ØØØ]    4:
[PE3]    5:         ;Name des Codesgments
[XC4]    6: code      segment byte public
[SW2]    7:         ;CS zeigt auf CODE
[EA2]    8:           assume cs:code
[ØØØ]    9:
[9Y4]   1Ø:         ;Tab-Funktion wird deklariert
[AL1]   11:           public tab
[ØØØ]   12:
[765]   13: spalte    equ byte ptr ss:[bp+4]
[8B5]   14: resultat equ dword ptr ss:[bp+6]
[ØØØ]   15:
[ZH6]   16:         ;Tab-Funktion muß NEAR-Prozedur sein
[3R2]   17: tab       proc  near
[B91]   18:           push  bp
[UH5]   19:         ;Übertrage Stack-Pointer nach BX
[8W1]   2Ø:           mov   bp,sp
[2S2]   21:         ;BIOS-Funktion 15
[AN1]   22:           mov   ah,15
[V83]   23:         ;Videostatus lesen
[G81]   24:           int   1Øh
[UM2]   25:         ;BIOS-Funktion 3
[WE1]   26:           mov   ah,3
[SK3]   27:         ;Cursorpositon lesen
[G81]   28:           int   1Øh
[Ø73]   29:         ;lade SPALTE nach AX
[9L2]   3Ø:           mov   al,spalte
[DM2]   31:         ;bilde Differenz
[GC4]   32:         ;SPALTE-AKTUELLE_SPALTE
[HS1]   33:           sub   al,dl
[1A2]   34:         ;ziehe eins ab
[L31]   35:           dec   al
[ES3]   36:         ;übertrage sie nach CL
[2U1]   37:           mov   cl,al
[WN2]   38:         ;und lösche CH
[UT1]   39:           xor   ch,ch
[F34]   4Ø:         ;lade Adresse des Strings
[2V1]   41:         ;nach ES:DI
[8U2]   42:           les   di,resultat
[ØØØ]   43:
[DS3]   44:         ;vorwärts speichern
[ULØ]   45:           cld
[WX3]   46:         ;speichere sie im String
[NXØ]   47:           stosb
[7H5]   48:         ;Leerzeichen sollen in den String
[3L1]   49:           mov   al,' '
[ZY5]   5Ø:         ;speichere die erforderliche Anzahl
[KX1]   51:           rep   stosb
[ØØØ]   52:
[571]   53:           pop   bp
[XN3]   54:         ;und räume Stapel auf
[B21]   55:           ret   2
[U13]   56:         ;Ende der Prozedur
[YR1]   57: tab       endp
[FØ3]   58:         ;Ende des Segments
[TT1]   59: code      ends
[UØ3]   6Ø:         ;Ende des Listings
[4MØ]   61:           end
```

Listing 1. Der Assemblerquelltext »tab.asm« für die Tab-Funktion

```
[7B5]    1: { Programmname: TABDEMO.PAS  }
[LG4]    2: { (C) 1989 Everts & Hagedorn }
[ZB4]    3: {           DOS International }
[ØØØ]    4:
[LV2]    5: program tabdemo;
[ØØØ]    6:
[DM3]    7: {$l c:\asm\tab.obj}
[PK5]    8: { Binde Maschinenprogramm ein }
[ØØØ]    9:
[DF6]   1Ø: { Benutzung erst ab Turbo Pascal 4.Ø }
[5H1]   11: uses crt;
[ØØØ]   12:
[W47]   13: { Die Assembler-Funktion wird definiert }
[W46]   14: function tab(spalte: byte): string;
[VL1]   15:                             external;
[ØØØ]   16:
[5P2]   17: var index: byte;
[ØØØ]   18:
[MUØ]   19: begin
[DB1]   2Ø:   clrscr;
[WB5]   21:   writeln('Dieser Text ist ohne '+
[L64]   22:           'Tabulator geschrieben.');
[XF1]   23:   writeln;
[793]   24:   for index:=1 to 12 do
[KU3]   25:     writeln(tab(5+index*2),
[DU3]   26:             'und dieser Text mit !')
[KNØ]   27: end.
```

Listing 2. »tabdemo.pas«: Ein Programmbeispiel für den Einsatz der Tab-Funktion

MaskEdit PLUS



Version 5.0

Professioneller Maskengenerator für Turbo Pascal 3/4/5, Turbo C ab V.1.5, dBase III + Microsoft C und Quick C, JPI TopSpeed Modula-2

Mit der neuen Version von MaskEdit Plus zaubern Sie komfortable Bildschirmmasken in Sekundenschnelle.

- Jetzt alle Funktionen über Pulldown-Menüs oder Hotkeys
- Jederzeit kontextbezogene Hilfe auf Tastendruck
- Ausnutzung des gesamten Bildschirms zur Maskenerstellung
- Unterstützung von zwei Bildschirmen gleichzeitig
- Leichte Konfiguration von Hotkeys und Pulldown-Menüs
- Verbesserte oder völlig neu entwickelte Maskentreiber

Professionell:

Mit über 50 Funktionen zur Maskenerstellung können auch Teile einer Maske kopiert, gelöscht, verschoben, zentriert, gefärbt und gefüllt werden.
Der komplette IBM-Zeichensatz bis 255 wird genauso wie die gleichzeitige Bearbeitung von zwei Masken und die Definition von Ein-/Ausgabefeldern unterstützt.
Mehrere Dateiausgabeformate wie ASCII, Binär, "Gepackt", dBase und ANSI (Farbige Masken auch mit dem DOS-Befehl TYPE!) sind im Lieferumfang enthalten.

Komfort:

Neben acht verschiedenen Rahmentypen sind acht oder 16 Hintergrundfarben, Schattenwurf und Cursorkoordinaten im Pascal- oder dBase-Format darstellbar.
Bei Rahmen- und Linienzeichnen werden die Kreuzungspunkte automatisch angepaßt. Eine integrierte ASCII-Tabelle und ein universeller Maustreiber machen Tempo.

Programmier-Highlights der Maskentreiber

Sämtliche Treiber werden als Sourcecode geliefert.
Eigene Routinen können zur Kontrolle von Eingaben, Hilfetexten etc. problemlos in den Steuerungsablauf eingebunden werden.
Integration von Datum, aktueller Uhrzeit und automatischer Typüberprüfung bei der Eingabe, auch für den zusätzlichen Datentyp DATUM.
Aufruf der generierten Masken durch einen einzigen Befehl und Steuerung der Ein-/Ausgabefelder durch den Maskentreiber.
Wahlweise Ausschluß von Teilmengen des Zeichensatzes und automatische Eingabe in Großbuchstaben inklusive der Umlaute.
Bei Änderungen von definierten Feldern kann die Kontrolle an das eigene Programm übergeben werden.
Jedes Eingabefeld kann einzeln gefärbt werden, ist voll editierbar und läßt sowohl rechts- wie auch linksbündige Ausgabe zu.
Der Cursor wird innerhalb der Eingabemaske durch die Pfeil- und TAB-Tasten bewegt, die Eingabemaske kann durch Funktionstasten-Betätigung verlassen werden.

MaskEdit Plus I Turbo Pascal 3/4/5 und dBase III+	**DM 99,– ***
MaskEdit Plus II Microsoft C und Quick C, Turbo C ab V.1.5 und dBase III+	**DM 99,– ***
MaskEdit Plus III JPI TopSpeed Modula-2 und dBase III+	**DM 99,– ***
Paketpreis mit zwei Treibern	**DM 149,– ***

Unabhängig von der Anzahl der bestellten Programme berechnen wir für das Inland 4,– DM bzw. für das Ausland 6,– DM Porto und Verpackung.

Bitte benutzen Sie die Bestellkarte.

DMV-Verlag · Postfach 250 · 3440 Eschwege

DMV Daten- und Medienverlag

Assembler und C als Duo Infernale

Gemischte Programmierung wird bei allen großen Software-Herstellern praktiziert. Unter Beachtung gewisser Regeln und mit der passenden Kombination von Assembler und Compiler ist es auch im nichtkommerziellen Bereich sehr effektiv, C-Quellcode mit Assembleranteilen zu versehen.

Jede Programmiersprache hat ihre Schwächen und Stärken. Warum also nicht die Schwächen einer Sprache durch die Stärken einer anderen kompensieren? Die Stärken von Assembler liegen in der Geschwindigkeit des lauffähigen Programms, im kompakten Maschinencode sowie in der Behandlung von Zeichenketten und »Far Pointern«. Dafür ist das Schreiben endloser Assemblerlistings zeitraubend und fehleranfällig. Auch die Konvertierung von Zahlen in Zeichen und umgekehrt ist nicht gerade eine Stärke von Assembler. Zu diesem Zweck bieten sich diverse C-Funktionen an. C ist ebenso interessant für kurzen Quellcode, wenn es sich nicht um zeitkritische Sequenzen handelt.

Fast jeder handelsübliche C-Compiler gestattet es, Assemblerzeilen in den Quellcode einzufügen. Diese sogenannte »Inline-Programmierung« bietet die größten Vorteile. Da beim Linken der Objektdateien zuerst das im Compiler-System enthaltene Startmodul gebunden wird, ersparen Sie sich das Schreiben vieler Assemblerzeilen. In diesem Startmodul sind nämlich alle Angaben enthalten, die Sie sonst in den Kopf eines Assemblermoduls schreiben müssen. Es sorgt für die richtigen Daten- und Code-Anweisungen, legt Speichermodell, Rücksprungadresse und ähnliches mehr fest. Beim »Inline-Programmieren« sind also nur wenige Regeln zu beachten.

Das folgende Beispiel zeigt ein C-Modul mit »Assembler-Inlinecode« in Turbo C 2.0:

```
int compare(int Var1, int Var2)
{
asm mov ax,Var1
/* Var1 in ax laden */
asm cmp ax,Var2
/* mit Var2 vergleichen */
asm jle C_Label
/* wenn kleiner zum C_Label */
asm mov ax,Var2
/* sonst Var2 in ax laden */
C_Label:
return(_AX);
/* Rückgabe kleinerer Wert */
}
```

Jede Assemblerzeile beginnt in Turbo C stets mit »asm«, bei MS-C ist es noch einfacher. Der folgende Teil unterscheidet sich von reinem C-Code nur dadurch, daß die Anweisung nicht mit einem Semikolon abgeschlossen werden muß. Ein Druck auf die Enter-Taste reicht aus. Da es sich um ein C-Modul handelt, sind auch nur C-Kommentare erlaubt. Von der Version 2.0 an erwartet »tcc.exe« den Borland-Assembler TASM als »Partner« beim Erzeugen der Objektdatei. Frühere Versionen arbeiten mit MASM zusammen. Aus der interaktiven Entwicklungsumgebung »tc.exe« können Sie das Modul nicht compilieren.

Bei Assembler-Inlinecode kümmert sich der Compiler um die Einzelheiten

Steht das Modul innerhalb eines längeren Listings, das geschlossen compiliert werden soll, sollte die erste Zeile im Kopf des Listings »#pragma inline« lauten. Damit erfährt der Compiler, daß Assembler im Quellcode vorhanden ist. Ein Aufruf von TCC mit dem Schalter »-B« hat die gleiche Wirkung. Das spart Zeit. Trifft der Compiler ohne vorherige Information auf Assemblerbefehle, startet er sich mit der Option »-B« neu.

Das Modul »compare()« ersetzt eine Verzweigung »if/else«, ist aber, trotz Parameterübergabe, schneller in der Ausführung; der erzeugte Maschinencode ist kompakter. Sein Einsatz lohnt sich bei zeitkritischen Routinen. Allerdings soll dies nur ein Beispiel dafür sein, wie leicht Assembler in C implementiert werden kann. Sie müssen nur auf die Werte der Variablen achten und die richtigen Register benutzen. Eine der beiden Variablen »Var1« oder »Var2« in »AH« oder »AL« laden zu wollen, würde unweigerlich zu einem Fehler führen. Ein kurzes Listing (Bild) zeigt, wie der Compiler das Modul in Assembler übersetzt:

Um dieses Listing zu erhalten, aus dem lediglich die Anweisungen an den Debugger entfernt sind, erfolgt von der MS-DOS-Kommandozeile der Aufruf »tcc -S compare«. Der Schalter »-S« weist den Compiler an, nur die Assemblerdatei zu erzeugen. In den Zeilen 17 bis 20 finden Sie den Assembler-Inlinecode wieder. Da die dem Modul übergebenen Variablen auf dem Stack liegen (Var1 = [bp + 4], Var2 = [bp + 6]), sorgt der Compiler auch für die richtige Adresse beim Laden in »AX«. Würde es sich um Long-Werte handeln, wäre Var2 = [bp + 8].

Speziell für C-Programmierer, die zwar den Aufbau der CPU und die »Mnemonics« kennen, sich aber noch keine reinen Assemblerprogramme

```
01:  _TEXT     segment       byte public 'CODE'
02:  DGROUP    group         _DATA,_BSS
03:            assume        cs:_TEXT,ds:DGROUP,ss:DGROUP
04: _TEXT      ends
05: _DATA      segment word public 'DATA'
06: d@         label         byte
07: d@w        label         word
08: _DATA      ends
09: _BSS       segment word public 'BSS'
10: b@         label         byte
11: b@w        label         word
12: _BSS       ends
13: _TEXT      segment       byte public 'CODE'
14: _compare   proc          near
15:                          push          bp
16:                          mov           bp,sp
17:                          mov           ax,[bp+4]
18:                          cmp           ax,[bp+6]
19:                          jle           @2
20:                          mov           ax,[bp+6]
21: @2:
22:                          jmp           short @1
23: @1:
24:                          pop           bp
25:                          ret
26: _compare   endp
27: _TEXT      ends
28: _DATA      segment word public 'DATA'
29: s@         label         byte
30: _DATA      ends
31: _TEXT      segment       byte public 'CODE'
32: _TEXT      ends
33:                          public        _compare
34:                          end
```

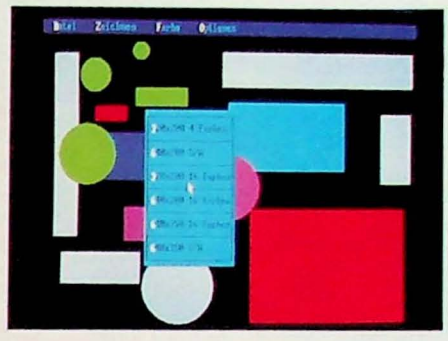
Turbo C erzeugt diesen Code; in den Zeilen 17 bis 20 findet sich der Assembler-Inlinecode wieder

DOS SOFTWARE TOOLS

JETZT NEU
Drei Toolboxen für GW-Basic

GW-Basic-Toolboxen

Warum kompliziert, wenn's doch so einfach geht! Dieser Wahlspruch gilt ab sofort auch für GW-Basic und SBasic, dem Compiler aus DOS-Extra 6/89. Verwenden Sie die neuen, leistungsfähigen GW-Basic-Toolboxen, und das Programmieren macht doppelt so viel Spaß. Trotz professioneller Leistung brauchen Sie auf Komfort nicht zu verzichten: Die Bedienung aller GW-Basic-Toolboxen gleicht der Bedienung des Basic-Interpreters "bis aufs Haar". Sie können problemlos mehrere GW-Basic-Toolboxen gleichzeitig laden und nutzen, denn es werden im Basic-Speicherbereich nicht mehr als zwei Bytes belegt. Mit jeder GW-Basic-Toolbox erhalten Sie ein komplettes, leistungsfähiges Beispielprogramm.



"Oberfläche"

Best.-Nr. 6009/6010

Mit dieser Toolbox realisieren Sie in GW-Basic eine komfortabel zu bedienende Benutzeroberfläche mit Pulldown-Menüs und kompletter Mausunterstützung.
Die Leistungsmerkmale:

- Vollständige Maussteuerung im Text- und Grafikmodus
- Spezielle Menüleisten für häufig benötigte Eingaben
- Popup-Menüs mit Meldungen bis zu fünf Zeilen Länge
- Position, Farben und Rahmen aller Menüs frei wählbar
- Unterstützung aller Bildschirmmodi von GW-Basic
- Inklusive einem fertigen Malprogramm als Beispiel

Jede Toolbox nur 69,– DM*

"DOS-Funktionen"

Best.-Nr. 6011/6012

Mit mehr als 30 DOS-Funktionen erlaubt Ihnen diese Toolbox von GW-Basic aus einen komfortablen Zugang zum MS-DOS-System. Die Leistungsmerkmale:

- Kompatibel zu MS-DOS von Version 2.1 bis 4.01
- Laufwerke wechseln, aktuelles Laufwerk ermitteln etc.
- Suchen nach Dateien mit den Wildcards "*" und "?"
- Beschleunigte Bildschirmausgabe über Systemfunktionen
- Zugang zu wichtigen, internen Daten des MS-DOS-Systems
- Benutzung des Druckprogramms Print von GW-Basic aus
- Inklusive Beispielprogramm: Eine komplette DOS-Shell

"Hercules-Grafik"

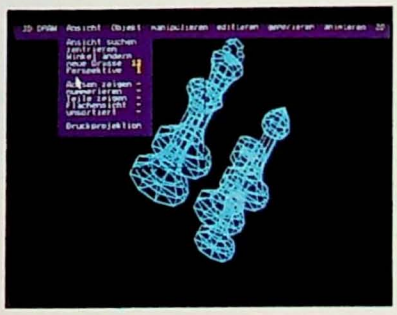
Best.-Nr. 6013/6014

Mit dieser Toolbox sind die Zeiten vorbei, in denen von GW-Basic nur der Textmodus der Hercules-Grafikkarte unterstützt wurde: Mit der GW-Basic-Toolbox »Hercules-Grafik« programmieren Sie unter GW-Basic nun auch im Grafikmodus.
Die Leistungsmerkmale:

- Routinen zum Ein- und Ausschalten des Grafikmodus
- Zwei verfügbare Grafikseiten mit Umschaltmöglichkeit
- Funktionen für Linie, Bogen, Kreis, Ellipse und Vieleck
- Ausfüllen von Flächen mit beliebigen Punktmustern
- Speichern und Restaurieren des Grafikbildschirms
- Setzen, Löschen und Abfragen von einzelnen Bildpunkten
- Emulation der GW-Basic-Befehle »View« und »Window«
- Inklusive Beispielprogramm: Präsentationsgrafik

3D-Draw

Das komfortable CAD-Programm »3D-Draw« bietet verdeckte Flächen und, als Besonderheit, Lichtquellensimulation. Mit der integrierten Animationsroutine erzeugen Sie beliebig lange Filme.
Die Leistungsmerkmale:



* Ansicht durch Eingabe der Projektionswinkel frei wählbar * Auf Wunsch lassen sich drei verschiedene 2D-Ansichten erzeugen, die am unteren Bildschirmrand eingeblendet werden * 2D-Editor zur schnellen Konstruktion von Objekten * Integrierter Taschenrechner für Konstruktionsdaten * Einstellbare Größe und perspektivische Verzerrung * Kombination von getrennt konstruierten Teilen * 3D-Funktion als Manipulationswerkzeug * Punkte, Kanten und Flächen sind frei editierbar * Drei Raumachsen mit 1600-Ebenen-Editor * Verdeckte Flächen * Lichtquellensimulator * Animationsroutine mit Größen- und Perspektivenveränderung von beliebiger Länge; ein eigener Animationslinker ist eingeschlossen * Verformen der Objekte durch Zufallszahl oder Verknüpfung mit mathematischen Formeln * Erzeugung von Rotations- und Translationskörpern und modifizierten Duplikaten * Programmierer-Schnittstelle, Objekte lassen sich mit einem Textprogramm oder Editor generieren * Vielzahl von Ausgabeformaten: 3D-Draw, Autocad-Script, Postscript, EPS-Format und HPGL-Plot-Format * Alle gängigen Grafikkarten werden unterstützt, komplette Ausnutzung der EGA-Palette mit 64 Farben

Systemvoraussetzungen:

Computer: PC, XT, AT oder Kompatibler mit mindestens 640 KByte Arbeitsspeicher, ein oder zwei Diskettenlaufwerke beziehungsweise ein Diskettenlaufwerk und eine Festplatte.

Grafikkarten: Hercules-, CGA-, EGA- oder VGA-Karte

Ausgabemedien: Epson-kompatibler Drucker, Ausgabe von Postscript-Dateien in Verbindung mit Postscript-Emulator, Postscript-fähiger Laserdrucker, Plotter mit HPGL-Standard.

Sonstiges: Microsoft-kompatible Maus unbedingt erforderlich, MS-DOS ab Version 3.0

nur 99,- DM*

Best.-Nr. 6007/6008

Resident-Software-Manager

Menügesteuertes Löschen und Nachladen von residenten TSR-Programmen löst Ihre Speicherplatzprobleme: RSM

* zeigt den noch freien Arbeitsspeicher * registriert geladene residente Programme * erinnert an die Aktivierungstastenkombination * informiert über benutzte Interrupt-Vektoren * löscht beliebige, speicherresidente Programme * lädt mehrere TSR-Programme in einem Arbeitsgang * bietet umfassende, komfortable Menüsteuerung * besitzt einen universellen Maustreiber * belegt lediglich 15 KByte Speicherplatz * informiert mit einem umfangreichen Handbuch

Für alle PCs, XTs, ATs und Kompatible mit mindestens 256 KByte Arbeitsspeicher

Best.-Nr. 6001/6002

nur 69,- DM*

PC-Utilities

nur 69,– DM*

Best.-Nr. 6003/6004

Softreset: Spannen Sie Ihre Nerven nicht weiterhin auf die Folter! Mit Softreset bleibt der komplette Speicherinhalt auch nach einem Reset erhalten und kann auf Diskette oder Festplatte gespeichert werden. **Zeitlupe PC:** Keine Angst vor Geschwindigkeit! Mit Zeitlupe PC verlangsamen Sie Ihren Computer bis zu einem Bruchteil der Taktfrequenz eines 4,77- MHz-PC. Mit Boot-Option! **Extras:** Der universell einsetzbare AT-Tastaturtreiber »TE« mit Quellcode. Passen Sie Ihre Tastatur an Ihre persönlichen Bedürfnisse an!

Call: verschachtelt Aufrufe von Batch-Dateien. **Code:** Paßwortverschlüsselung. **Makeroom:** löscht Dateien gemäß Suffix. **Setdir, Restdir:** aktivieren nach Programmaufruf das Original-Verzeichnis. **Ty:** Hex-Dump, Binärformat oder Zeilennummer auf dem Drucker oder Bildschirm aus. **Extras:** »SCopy«, das professionelle Kopierprogramm, »Filed«, der Editor, dem nichts verborgen bleibt.

MS-DOS-Utilities

nur 49,– DM*

Best.-Nr. 6005/6006

* Sämtliche Programme sind wahlweise auf 5 1/4-Zoll- oder 3 1/2-Zoll-Diskette erhältlich. (Bitte bei Bestellung unbedingt angeben.) Unabhängig von der Anzahl der bestellten Artikel berechnen wir für das Inland 4,– DM bzw. für das Ausland 6,– DM Porto und Verpackung. Unverbindliche Preisempfehlung. Bitte benutzen Sie die Bestellkarte.

DMV-Verlag · Abteilung DOS-Software · Postfach 250 · 3440 Eschwege

DMV Daten- und Medienverlag

zutrauen, ist diese Art der gemischten Programmierung nahezu ideal. So schreiben Sie den gesamten Quellcode erst in C, beseitigen die Fehler und implementieren Assemblerbefehle, wo es erforderlich scheint. Allerdings müssen Sie dabei einige Regeln beachten. Werden im C-Modul Registervariablen verwendet, dürfen die Assemblerzeilen die Register »SI« und »DI« nicht benutzen, da in ihnen die Variablen gespeichert sind. Außerdem erwartet jedes C-Modul beim Einsprung ein intaktes Register »BP«. Wie Sie aus dem Listing ersehen, wird »BP« auch in Zeile 15 auf den Stack gerettet, bevor der Stackpointer »SP« in Zeile 16 in das Register geladen wird. In Zeile 24, noch vor dem Rücksprung in Zeile 25, stellt der Compiler den Originalzustand wieder her.

Wenn Sie mit dem Befehlssatz des 80x87 vertraut sind, können Sie auch diese Befehle implementieren. Programmieren Sie die regulären Register »ST« bis »ST(7)« des Arithmetikprozessors und binden Sie beim Linken die Bibliothek »emu.lib« mit ein, wenn Ihr Rechner keinen Arithmetikprozessor besitzt. Der Compiler erzeugt dann einen »statischen Stack« mit der Registerstruktur eines Arithmetikprozessors.

Sehr gut lassen sich auch die vielfältigen Sprungbefehle, über die die Assemblersprache verfügt, für Assembler-Inlinecode ausnutzen. Die Tabelle listet diese Befehle auf. Dabei müssen Sie nur darauf achten, daß es sich immer um »kurze« Sprünge handelt, also das Sprunglabel innerhalb einer Entfernung von 128 Byte liegt.

Bedingungsabhängige Speicherreservierungen, unformatierte String-Ein- und Ausgaben, Schreiben in bestimmte Speicherbereiche und ähnliches eignet sich ebenfalls für Assembler-Inlinecode. Hier wird der Geschwindigkeitsvorteil von Assembler genutzt und der kompakte Maschinencode, den TASM oder MASM erzeugen. Sie sollten dann die »Loop-Befehle« einsetzen, die das CX-Register automatisch als Zähler verwenden und schneller arbeiten als entsprechende Jump-Befehle. Kurz, alle zeitkritischen For- und While-Schleifen sind effizient durch Assembler-Inlinecode ersetzbar.

Doch nicht nur innerhalb von Modulen können Sie Assembler-Inlinecode implementieren. Die Zeile

```
asm INCLUDE ASMMODUL.ASM
```

irgendwo im C-Quelltext, außerhalb eines Moduls, weist TCC/TASM an, an dieser Stelle das Assemblermodul »asmmodul.asm« in die zu erzeugende Asm/Obj-Datei einzufügen. Ist dieses Modul in sich abgeschlossen, kann es wie jedes andere C-Modul im Quellcode aufgerufen werden. In diesem Fall ist nur zu beachten, daß die Angabe des Speichermodells im Assemblermodul identisch mit dem Modell ist, mit dem »tcc.exe« compiliert.

Wenn Sie bereits über gute Assemblermodule oder Makros verfügen, steht ihrer Verwendung in C nichts im Wege. Die direkte Implementierung ist häufig einfacher als das spätere Einbinden durch den Linker. Speziell in umfangreichen Programmen gewinnen sie damit einen gewissen Einfluß auf die Ablage der Module im Maschinencode und können segmentübergreifende Sprünge verringern oder gar vermeiden. Assemblerbefehle innerhalb eines C-Moduls gelangen beim Assemblieren in das Codesegment von Turbo C, Assemblerbefehle von außerhalb dagegen in das Datensegment.

Um C-Module und Assemblermodule zu einem lauffähigen Programm zu linken, müssen einige Voraussetzungen erfüllt sein. Die Speichermodelle müssen identisch sein und die Segmentbezeichnungen beider Modularten demselben Schema entsprechen. Diese Forderungen werden mit den Deklarationen von »DOSSEG« und ».MODEL ???« im Kopf des Assemblermoduls erfüllt. Funktions- und Variablennamen müssen übereinstimmen, und zum Binden sich eventuell unterscheidender Objektdateien muß »tlink« benutzt werden. Bei den Microsoft-Produkten MS-C, MASM und dem Linker »link« gelten äquivalente Bedingungen.

Der Aufruf einer C-Funktion, hier »strcpy (Zielstring, Quellstring)«, aus einem Assemblermodul sieht dementsprechend folgendermaßen aus:

```
DOSSEG
.MODEL SMALL
.
.
les ax,Quellstring ; letzter Parameter
push ax ; auf den Stack
lea ax,Zielstring ; Parameter davor
push ax ; auf den Stack call _strcpy ;
Aufruf C-Funktion
add sp,4 ; Stack abräumen
```

Die Parameterübergabe erfolgt nach C-Konvention von rechts nach links, dem Funktionsnamen wird ein Unterstrich vorangestellt und der Stack wird nach dem Aufruf wieder abgeräumt.

Gemischte Programmierung bringt nicht nur Vorteile mit sich. Einige Nachteile müssen Sie dabei auch in Kauf nehmen. So geht eine der Stärken von C verloren – die Portabilität auf andere Systeme. Assembler für den 80x86 ist naturgemäß auch nur für diese Prozessorfamilie bestimmt und wird zum Beispiel nie unter dem Atari- oder Macintosh-Betriebssystem laufen, die auf dem Motorola- Prozessor 680x0 aufgebaut sind. Die Wartung des Quellcodes wird erschwert. Programmierer, die Assembler nicht beherrschen, verstehen den Sinn dieser Codierung nicht. Es empfiehlt sich deshalb, Assemblersequenzen gut zu kommentieren. Wer es gewohnt ist, mit »tc.exe« zu arbeiten, der interaktiven Oberfläche von Turbo C, muß sich erst mit »tcc« und »tlink« vertraut machen, bevor er ablauffähige Programme mit Assembleranteilen erzeugen kann.

Turbo C verliert durch Assembler-Inlinecode außerdem einige Möglichkeiten der Sprungoptimierung und kann Registervariablen nicht einsetzen, wenn »SI« und »DI« benutzt werden, was bei fast allen String-Operationen der Fall ist. Die Fehlersuche wird erschwert, da sich Zeilennummern auf die Asm-Datei beziehen, die temporär angelegt wird, und nicht auf den C-Quellcode.

Zusammenfassend läßt sich sagen, daß sich Assembler-Inlinecode zwar sehr leicht schreiben läßt, es andererseits aber häufig sinnvoller ist, komplette Assemblermodule in C zu implementieren.

(Jürgen Hochstädt/mk)

Bedingte Assemblersprungbefehle

Befehl	Sprung:	geprüfte Flags
JAE/JNB	wenn größer gleich oder nicht kleiner	CF = 0
JB/JNAE	wenn kleiner oder nicht größer gleich	CF = 1
JBE/JNA	wenn kleiner gleich oder nicht größer	CF = 1 oder ZF = 1
JA/JNBE	wenn größer oder nicht kleiner gleich	CF = 0 oder ZF = 0
JE/JZ	wenn beide gleich sind	ZF = 1
JNE/JNZ	wenn Operanden ungleich	ZF = 0
JL/JNGE	(mit Vorzeichen) wenn kleiner oder nicht größer gleich	SF <> OF
JGE/JNL	(mit Vorzeichen) wenn größer gleich oder nicht kleiner	SF <> OF
JLE/JNG	(mit Vorzeichen) wenn kleiner gleich oder nicht größer	SF <> OF oder ZF = 1
JG/JNLE	(mit Vorzeichen) wenn größer oder nicht kleiner gleich	CF = 0 und ZF = 0
JP/JPE	falls Parität	PF = 1
JNP/JPO	falls keine Parität	PF = 0
JS	falls Vorzeichen gesetzt	SF = 1
JNS	falls kein Vorzeichen	SF = 0
JC	falls Carry gesetzt	CF = 1
JNC	falls kein Carry	CF = 0
JO	falls Überlauf	OF = 1
JNO	falls kein Überlauf	OF = 0

Flags: CF = Carry, SF = Vorzeichen, OF = Überlauf, PF = Parität

DOS-EXTRA
DATABOX

– Die Software zur DOS-EXTRA Nr. 8 –

auf 2 Disketten im Format 5 1/4“ oder
1 Diskette im Format 3 1/2“

sagenhaft günstig, extra viel gute Software fürs Geld!

DATABOX:

- mehr als der übliche Softwareservice
- bringt alle Programme der DOS EXTRA auf Datenträger

FFür alle PCs unter MS-DOS als 5 1/4“- coder 3 1/2“-Diskette

DDen Aufbau der Databox-Diskette finden SSie in der Datei »read.me« erklärt. Geben SSie deshalb vor dem Start »type read.me« eein.
SSämtliche Pascal- und C-Programme liegen auch in einer compilierten Form für HHercules- und Farbgrafikkarten vor. Es ist aallerdings nicht gewährleistet, daß diese VVersionen auf Hardware, die nicht vollständig kompatibel zum IBM PC ist, problemlos llaufen. In diesem Fall müssen Sie den QQuellcode nach der Anpassung mit dem eentsprechenden Compiler neu übersetzen.
DDie Basic-Programme arbeiten mit dem GW-Basic-Interpreter von Microsoft einwandfrei. Bei Interpretern und Compilern anderer Hersteller kann es zu Problemen bei der Programmausführung kommen, sofern diese nicht voll kompatibel sind.

Databoxinhalte der DOS EXTRA Nr. 8

- Makroassembler SAM III mit vielen Beispielen
- Exe2bin-Ersatz durch “exe2com“
- Programm zur Verbesserung der Speicherverwaltung unter MS-DOS
- Routine zum Speichern von Hardcopies auf Datenträger
- Alle Listings zum Einsteigerschwerpunkt
- SCheck 2.01 Checksummer für geprüfte Eingaben
- sowie alle begleitenden Listings zu den Werkstätten

2 Disketten 5 1/4“ (Bestell-Nr. 3819)
1 Diskette 3 1/2“ (Bestell-Nr. 3820)

DM 35,– *

Noch erhältlich:

Databox DOS EXTRA Nr. 1

- SAM, der DOS-Assembler
- Clock, Uhr-Utilities
- Pascal – Include-Datei zum Aufruf von BIOS-Funktionen unter Turbo Pascal
- Tips und Tricks für jeden Zweck

Diskette 5 1/4“
Diskette 3 1/2“ **DM 24,– ***

Databox DOS EXTRA Nr. 2

- DOS-Commander, eine Benutzeroberfläche für MS-DOS
- SCopy, ein Kopierprogramm
- SForth 3.1, ein Compiler im Selbstbau
- SCAD, (3D-Grafik)
- Hurrican, ein Actionspiel
- Supermind, ein Strategiespiel
- Utilities, Routinen zur täglichen Arbeit
- SCheck – Checksummer zur Eingabe

4 Disketten 5 1/4“
2 Disketten 3 1/2“ **DM 45,– ***

Databox DOS EXTRA Nr. 3

- Datenfernübertragungsprogramm in 100% Assemblercode
- Hercules-Grafik-Toolbox in Turbo Pascal
- Grafikfunktionen für Turbo C
- Zeichensatzeditor für Epsonkompatible Drucker
- Treiber für AT-Tastatur
- Alle begleitenden Listings zum Kurs »PC-glasklar«
- Demo für Sprachdigitalisierung

Databox DOS EXTRA Nr. 4

- Desktop-Publishing-Programm »SDTP«
- Kommandoeditor für MS-DOS
- Editor für Disketten und Festplatten
- Speicherresidente Programme auflisten und entfernen
- Paßwort-Schutz für MS-DOS
- Tastatureingabe automatisiert
- Clock-Utility für alle Grafikkarten
- Cursor und Zeichensatz für EGA-Karte
- Fenstertechnik für Turbo C
- Generator für Hilfsbildschirme
- Geprüfte Eingabe mit »SCheck«

Databox DOS EXTRA Nr. 5

- 16 Farben für die CGA-Farbgrafikkarte
- Snapshot: Bildschirmfotos per Software
- Druckerzeichensatz selbstgemacht
- Mausgesteuertes Zeichenprogramm
- Editor in GW-Basic
- Setup-Programm für ATs
- Maschinensprache für GW-Basic
- Erklärtes Listing: das Basic-Spiel »Stardos«
- Neuer Checksummer

jeweils
2 Disketten 5 1/4“
1 Diskette 3 1/2“

DM 35,– *

Databox DOS EXTRA Nr. 6

- SBasic-Compiler im Quelltext
- Datenbankprogramm »Parat«
- Fensterorientierte Funktionen für GW-Basic und SBasic
- Funktionen zur Stringverwaltung für GW-Basic und SBasic
- Turbo-Basic-Bibliothek für 16 Farben auf der CGA-Karte
- Hilfsfunktion für Basic-Grafikbefehle
- Programm zur Analyse von Grafikkarten
- Preview-Funktion für Dateien
- Maskengenerator zum Erzeugen von Binärdateien
- dBase-Datenbanken in Basic weiterverarbeiten
- Beispiele für rekursive Programmierung
- Superhirn für Quick Basic 4.0
- SCheck 2.0: Checksummer für geprüfte Eingaben in Basic
- sowie alle begleitenden Listings zu den Rubriken Basic intern und Tips & Tricks

Databox DOS EXTRA Nr. 7

- MS-DOS-Benutzeroberfläche Easydos
- Benutzeroberfläche und Präprozessor für SBasic
- Datengenerator für Basic, C und Pascal
- Sprite-Editor für Farbgrafikkarten
- DOS-Packer
- Residenten Programmen auf der Spur
- Virenschutzprogramm
- Monitorprogramm zur Tastaturauswertung
- Verwaltung von Programm-Modulen in C
- Weltkarte im Computer
- Schneller Zugriff auf der CGA-Karte
- Kurven einfacher plotten
- Einfacher Prozessor-Emulator
- SCheck 2.01: Checksummer für geprüfte Eingaben
- sowie alle begleitenden Listings zu den Grundlagenbeiträgen und der Tips-&-Tricks-Rubrik

* Unabhängig von der Anzahl der bestellten Programme berechnen wir für das Inland 4,– DM bzw. für das Ausland 6,– DM Porto und Verpackung. – Unverbindliche Preisempfehlung – Bitte benutzen Sie die Bestellkarte.

DMV-Verlag · Postfach 250 · 3440 Eschwege

dBase-Programme »tunen« mit Assemblermodulen

Selbst angesichts einer perfekt programmierten dBase-Anwendung (gleichgültig, mit welcher dBase-Version sie geschrieben wurde) bleiben oft noch Wünsche hinsichtlich der Geschwindigkeit beziehungsweise des Befehlsumfangs offen. In solchen Fällen helfen Ihnen Assemblerroutinen, also kleine Maschinenprogramme, die von dBase aus aufgerufen werden.

Assembler steht für Geschwindigkeit, das ist eine allgemein bekannte Tatsache. Auch daß sich Programmiersprachen wie Pascal, Basic oder C mit speziellen Assembler-Routinen »aufpeppen« lassen, ist kein Geheimnis mehr. Wie steht es nun aber mit dBase-Programmen – denn auch wenn man in dBase programmieren kann, versteht sich diese Datenbanksprache auch mit Assembler? Kurz und gut, auch Ihre dBase-Anwendung hat nichts gegen eine »Geschwindigkeitsspritze« durch einige kleine Maschinenprogramme.

Das Schema von Assembler-Programmen und dBase-Assemblermodul ist identisch

Um Assemblerroutinen (im folgenden auch Module genannt) anzufertigen, benötigen Sie einen Assembler, mit dem Sie die Listings mit der Endung »asm« sowie die MS-DOS-Programme »link.exe« und »exe2bin.com« eingeben. Für Leser, die über keinen Assembler verfügen, sind die Listings in diesem Beitrag auch als Hexdumps (Hexadezimalausdruck einer Datei) mit der Erweiterung »hex« abgedruckt. Diese Listings schreiben Sie mit einem Texteditor ab und wandeln die Datei (zum Beispiel »ton.hex«) über das Checksummen-Programm SCheck (das erste Listing in dieser Ausgabe) mit der Option »/b« in eine Binär-Datei um, die Sie später von dBase aus aufrufen können.

Der Aufbau eines dBase-Assemblermoduls entspricht grundsätzlich dem Schema von Assemblerprogrammen für eine Com-Datei. dBase verlangt allerdings einige Einschränkungen:

- Es dürfen höchstens fünf Module gleichzeitig geladen sein. Notfalls sind nicht benötigte Module mit »release..« zu entfernen.
- Die maximale Länge je Modul beträgt 32 KByte.
- In der Datei »config.db« muß »maxmem« auf einen Wert gesetzt werden, der um die Gesamtgröße der geladenen Module größer als 320 KByte ist, damit ein mit »run« aufgerufenes Programm die Module nicht überschreibt.
- Daten, die das Assemblerprogramm bearbeiten soll, müssen sich innerhalb der Programmdatei befinden. Sie müssen also für alle Variablen und Konstanten im Modul Platz reservieren. Für einen kompletten Bildschirminhalt sind das beispielsweise zwei KByte.
- Ein Argument dient Steuerzwecken im Modul. Es darf bezüglich der Länge nicht verändert werden. Argumente werden auf einem von dBase eingerichteten und verwalteten Stapel abgelegt und nach Ausführung des Moduls wieder zurückgeholt. Nur eine Zeichenkette oder eine Variable ist als Argument zulässig.
- Die Rückkehr zu dBase muß mit »far ret« erfolgen. Das erreichen Sie, indem Sie das Modul als »proc far« definieren. Wichtig ist, daß Sie das Modul mit »ret« und nicht mit einer MS-DOS-Funktion beenden.
- Am Programmanfang darf der sonst für Com-Dateien eingesetzte Pseudobefehl »org 100H« nicht verwendet werden.

Assembler macht's möglich: Tempo für dBase-Programme

Ein Assemblermodul wird assembliert, mit »link dateiname« gebunden (die Meldung wegen des fehlenden Stacks kann aufgrund des Com-Datei-kompatiblen Aufbaus ignoriert werden) und mit »exe2bin dateiname« in »dateiname.bin« umgewandelt. Bei allen Bearbeitungen geben Sie keine Erweiterung und beim Umwandeln auch keinen zweiten Dateinamen an. Das so angefertigte Programm laden Sie mit

```
load dateiname
```

in Ihr dBase-Programm. Die Erweiterung »bin« dürfen Sie weglassen, nicht aber »com« bei Verwendung von Com-Dateien. Der Aufruf erfolgt mit

```
call dateiname
```

Für verschiedene Situationen sieht dBase leider nicht auch verschiedene Töne vor. Nutzen Sie deshalb das Assemblermodul für eine erweiterte Tonausgabe in Listing 1 und 2. Das Modul bietet Ihnen die Möglichkeit, beim Aufruf der Routine auch ein Argument zu übergeben, durch das die gewünschte Tonfolge ausgewählt wird. Durch Erweiterungen an diesem Modul und durch die Übergabe zusätzlicher Argumente in Verbindung mit einer Fehlerbehandlungsroutine können Sie das Programm so erweitern, daß der Benutzer bereits am Ton erkennt, was er falsch gemacht hat.

Spiel mir den richtigen Ton, und ich sage Dir, was Dir fehlt

Die Frequenz in Hertz wird dabei durch einen Zählerwert im Register CX bestimmt, der sich wie folgt berechnen läßt:

```
cx = 1000000 : (7,14 x f)
```

Die Tondauer ergibt sich aus einem Zählerwert im Register DX:

```
dx = f x Zeit in Sekunden
```

»ton.asm« erzeugt einen Ton mit 3072 Hz (entspricht dem »viergestrichenen g«) von einer Achtelsekunde Dauer. Die Argumente werden in einer Zeichenkette oder einem alphanumerischen Feld auf einen Stapel gelegt. Dabei zeigt das Registerpaar DS:BX auf das erste Zeichen, das heißt, es kann mit dem Befehl

```
mov ah,ds:[bx]
```

gelesen werden. Die weiteren Zeichen des Arguments werden durch entsprechende Wiederholung und Inkrementierung von BX oder mit »rep movsb« beziehungsweise »rep movsw« gelesen. Diese Befehle fügen Sie vor der Gleichsetzung der Register CS, DS und ES ein.

Anschließend vergleicht das Programm die Zeichen mit vorgegebenen Werten und verzweigt je nach Ergebnis in die entsprechenden Unterprogramme.

Nach dem Starten von »ton.prg« hören Sie zunächst den Einzelton, dann den Zweiklang. Dieses Modul können Sie zum Beispiel dazu verwenden, um auf verschiedene Fehler hinzuweisen. Wenn ein ungültiges Argument übergeben wird, erhalten Sie

eine Fehlermeldung. Das Modul gibt bei Eingabe eines falschen Arguments einen Buchstaben an dBase zurück (bei diesem Modul »X«). Dazu wird nach dem Vergleichen eine Verzweigung zu einem zweiten Schlußteil durchgeführt. Wichtig ist, daß zunächst die Originalwerte von BX und DS mit »pop« zurückgeholt werden und dann die Fehlermeldung über das AL-Register auf den Stapel geschrieben wird. Diese wird dBase wieder mit dem Registerpaar DS:BX übergeben und kann anschließend im dBase-Programm abgefragt werden. Verwenden Sie zum Testen des Moduls das Programm »ton.prg« (Listing 3).

Aus sinnlosen Zahlen, mit denen Sie nichts anfangen können, wird eine klare, aussagekräftige Meldung

Mit Assembler-Routinen lassen sich noch weitere nützliche Tricks in ein dBase-Programm einbauen. Als dBase-Anwender kennen Sie sicherlich die kryptische Ausgabe einer Fehlermeldung auf dem Bildschirm. Doch sehr viel Aussagekraft hat solch eine Zahl nicht; es wäre doch bedeutend einfacher, wenn im Klartext da stehen würde, welcher Art der Fehler ist, der das Programm daran hindert weiterzuarbeiten. Auch hier schafft ein kleines Assembler-Modul Abhilfe.

Das Modul »wechsel« (Listing 4 und 5) vertauscht die übergebene Zeichenkette mit einer im Modul gespeicherten. Eine Zahl als Fehlermeldung hilft dem Anwender nicht viel, er möchte am Bildschirm auch lesen, was nicht in Ordnung ist. Hierfür wird das Modul »wechsel« eingesetzt. Erhält das Modul zum Beispiel als Argument die Zeichenkette für einen bestimmten Fehler, so gibt es statt dessen eine dort abgelegte Meldung zurück. Durch entsprechende Erweiterungen können Abfragen und Sprungadressen eingebaut werden, nach dem Prinzip: »Wenn die Fehlernummer = xxx ist, dann gebe die Meldung »yyyyyyyy« an dBase zurück«.

Beachten Sie jedoch, daß am Programmbeginn das DS-Register erst mit dem CS-Register gleichgesetzt werden darf, wenn der Argument-String übernommen worden ist. Entsprechend muß am Programmende vor dem Zurückholen der Registerursprungswerte jeweils der Wert von DS und BX zur Bestimmung der Zieladresse für den Movsb-Befehl in ES und DI geladen werden. Mit solchen Modulen können Argumente an das Modul übergeben, dort bearbeitet und wieder an dBase zurückgegeben werden. Starten Sie dazu »wechsel:prg«.

Stellen Sie sich vor, Sie sind gerade damit beschäftigt, etwas einzugeben, und mittendrin klingelt das Telefon. Das Gespräch dauert länger, und währenddessen leuchtet Ihr Bildschirm weiter, verbraucht sinnlos Strom und brennt den Bildinhalt in die Mattscheibe ein. Dagegen hilft das dritte Modul, das den Bildschirm ab- und anschaltet (Listings 7 und 8). Zusammen mit einem entsprechenden dBase-Programm (Listing 9) kann es dazu verwendet werden, den Bildschirm beim Warten auf eine Eingabe von der Tastatur nach einer bestimmten Zeit abzuschalten und auf Tastendruck wieder einzuschalten. Ihr Monitor wird es Ihnen danken, wenn Sie ihn auf diese Weise entlasten.

Schalten Sie doch einfach (den Bildschirm) ab, wenn gerade einmal nichts zu tun ist

Sie erkennen an den drei hier vorgestellten Assemblermodulen, daß es oft sehr nützlich sein kann, auf andere Programmiersprachen auszuweichen, um in dBase Funktionen zu realisieren, die eigentlich gar nicht möglich sind, beziehungsweise in dBase viel zu langsam wären.

(Heinrich Baumann/hw)

```
[WXA]    1: ;ton piepst mit einem kurzen oder zwei vers
            chieden hohen Tönen
[8K8]    2: ;als Parameter wird der Buchstabe E oder D
            übergeben
[P3Ø]    3: ;
[CA4]    4: code segment public 'code'
[PB3]    5:      ;Start des Segments
[T32]    6: main proc far
[Z93]    7:      ;DBase verlangt far
[3U4]    8: assume cs:code,ds:code,es:code
[P3Ø]    9: ;
[CD3]   1Ø: START:    jmp    ANFANG
[P3Ø]   11: ;
[562]   12:      ;DATENBEREICH
[P3Ø]   13: ;
[AC5]   14: FEHLM db 'UNGÜLTIGER PARAMETER'
[4F2]   15:      ;Fehlermeldung
[P3Ø]   16: ;
[CP2]   17: ANFANG: push es
[KM3]   18:      ;Sichern der Register
[SD1]   19:      push ss
[BC1]   2Ø:      push ds
[TC1]   21:      push bx
[P3Ø]   22: ;
[KS2]   23:      mov  ah,ds:[bx]
[ØY2]   24:      ;Parameter nach ah
[8C1]   25:      push cs
[771]   26:      ;ds = cs
[5A1]   27:      pop  ds
[8C1]   28:      push cs
[A71]   29:      ;es = cs
[8A1]   3Ø:      pop  es
[P3Ø]   31: ;
[XF1]   32:      cmp ah,'E'
[G43]   33:      ;Ist Parameter = E?
[VK1]   34:      je EINFACH
[PK5]   35:      ;Wenn ja verzweige nach Einzelton
[VF1]   36:      cmp ah,'D'
[D43]   37:      ;Ist Parameter = D?
[KE1]   38:      je DOPPEL
[4L5]   39:      ;Wenn ja verzweige nach Doppelton
[2T1]   4Ø:      jmp FEHLER
[394]   41:      ;Weder E noch D ist Fehler
[P3Ø]   42: ;
[712]   43:      ;EINFACHTON
```

```
[P3Ø]   44: ;
[863]   45: EINFACH: call TONP1
[552]   46:      jmp SCHLUSS
[P3Ø]   47: ;
[9U1]   48:      ;DOPPELTON
[P3Ø]   49: ;
[W13]   5Ø: DOPPEL: call TONP2
[552]   51:      jmp SCHLUSS
[P3Ø]   52: ;
[H23]   53: FEHLER: call ERROR
[P3Ø]   54: ;
[6T2]   55: SCHLUSS: pop bx
[WØ4]   56:      ;Abschluss des Programms
[E31]   57:      pop ds
[C41]   58:      pop ss
[G31]   59:      pop es
[3PØ]   6Ø:      ret
[NB3]   61:      ;Rückkehr zu dBase
[P3Ø]   62: ;
[SM2]   63:      ;TONPROZEDUREN
[P3Ø]   64: ;
[712]   65:      ;EINFACHTON
[P3Ø]   66: ;
[VH2]   67: TONP1 proc near
[EMØ]   68:      cli
[CL1]   69:      mov dx,35Ø
[B83]   7Ø:      ;Tondauer ca Ø,125 SEC
[2D1]   71:      in  al,61H
[XU1]   72:      and al,ØFEH
[M82]   73: AGAIN1: or al,2
[YN1]   74:      out 61H,al
[XE1]   75:      mov cx,46
[723]   76:      ;Frequenz ca 3Ø72 Hz
[VR2]   77: WAIT1: loop WAIT1
[UU1]   78:      and al,ØFDH
[YN1]   79:      out 61H,al
[XE1]   8Ø:      mov cx,46
[ZR2]   81: WAIT2: loop WAIT2
[LØ1]   82:      dec dx
[5F3]   83:      ;Ende eines Impulses
[TP1]   84:      jnz AGAIN1
[9L3]   85:      ;Wiederhole bis dx = Ø
[6PØ]   86:      sti
[3PØ]   87:      ret
[TN1]   88:      tonp1 endp
```

```
[P3Ø]   89: ;
[YS3]   9Ø:     ;DOPPELTON KLEINE TERZ
[P3Ø]   91: ;
[XH2]   92: TONP2 proc near
[EMØ]   93:     cli
[CL1]   94:     mov dx,35Ø
[XS3]   95:     ;erster Ton ca Ø,125 sec
[K71]   96:     in al,61H
[XU1]   97:     and al,ØFEH
[R82]   98: AGAIN2: or al,2
[YN1]   99:     out 61H,al
[XE1]  1ØØ:     mov cx,46
[723]  1Ø1:     ;Frequenz ca 3Ø72 Hz
[3S2]  1Ø2: WAIT3: loop WAIT3
[UU1]  1Ø3:     and al,ØFDH
[YN1]  1Ø4:     out 61H,al
[XE1]  1Ø5:     mov cx,46
[7S2]  1Ø6: WAIT4: loop WAIT4
[LØ1]  1Ø7:     dec dx
[5F3]  1Ø8:     ;Ende eines Impulses
[WP1]  1Ø9:     jnz AGAIN2
[P3Ø]  11Ø: ;
[6L1]  111:     mov dx,32Ø
[7Y3]  112:     ;zweiter Ton ca Ø,125 sec
[K71]  113:     in al,61H
[XU1]  114:     and al,ØFEH
[U82]  115: AGAIN3: or al,2
[YN1]  116:     out 61H,al
[WE1]  117:     mov cx,55
[L23]  118:     ;Frequenz ca 256Ø Hz
[BS2]  119: WAIT5: loop WAIT5
[UU1]  12Ø:     and al,ØFDH
[YN1]  121:     out 61H,al
[WE1]  122:     mov cx,55
[FS2]  123: WAIT6: loop WAIT6
[LØ1]  124:     dec dx
[ZP1]  125:     jnz AGAIN3
[P3Ø]  126: ;
[6PØ]  127:     sti
[3PØ]  128:     ret
[VN1]  129:     tonp2 endp
[P3Ø]  13Ø: ;
[RL2]  131: ERROR proc near
[YC4]  132:     ;Ausgabe der Fehlermeldung
[VB3]  133:     mov si,offset FEHLM
[XZ1]  134:     mov ax,ØBØØØH
[BC4]  135:     ;Start Bildschirmspeicher
[3H1]  136:     mov es,ax
[9N6]  137:     ;Beginn der Meldung bei Zeile 1Ø Platz
         3Ø
[FV1]  138:     mov    di,166Ø
[KX1]  139:     mov    ah,ØFØH
[Z34]  14Ø:     ;Umkehranzeige blinkend
[EM1]  141:     mov    cx,2Ø
[Z82]  142:     ;2Ø Buchstaben
[ULØ]  143:     cld
[G22]  144: AGAIN4: lodsb
[ZYØ]  145:     stosw
[XX1]  146:     loop AGAIN4
[3PØ]  147:     ret
[NR1]  148: error endp
[P3Ø]  149: ;
[UF1]  15Ø: main endp
[TF1]  151: code ends
[4K1]  152: end START
```

Listing 1. Das Modul »ton.asm« dient der gesteuerten Erzeugung mehrerer Töne

```
[RH5]  1: Ø6 16 1E 53 8A 27 ØE 1F ØE Ø7 8Ø FC 45 74 Ø8 8Ø
[VL5]  2: FC 44 74 Ø9 EB ØF 9Ø E8 14 ØØ EB Ø4 9Ø E8 2D ØØ
[GR5]  3: 5B 1F 17 Ø7 CB 5B 1F B4 58 88 27 17 Ø7 CB FA BA
[FM5]  4: 5E Ø1 E4 61 24 FE ØC Ø2 E6 61 B9 2E ØØ E2 FE 24
[A36]  5: FD E6 61 B9 2E ØØ E2 FE 4A 75 EB FB C3 FA BA 5E
[RL5]  6: Ø1 E4 61 24 FE ØC Ø2 E6 61 B9 2E ØØ E2 FE 24 FD
[FN5]  7: E6 61 B9 2E ØØ E2 FE 4A 75 EB BA 4Ø Ø1 E4 61 24
[SP5]  8: FE ØC Ø2 E6 61 B9 37 ØØ E2 FE 24 FD E6 61 B9 37
[VX2]  9: ØØ E2 FE 4A 75 EB FB C3
```

Listing 2. »ton.hex« ist der Hexadezimalausdruck von »ton.bin«

```
[ETØ]    1: *ton*
[P22]    2: set talk off
[3VØ]    3: clear
[LC1]    4: load ton
[Z41]    5: arg = "E"
[P12]    6: ? "Einzelton"
[WS2]    7: call ton with arg
[9RØ]    8: wait
[Y41]    9: arg = "D"
[AY1]   1Ø: ? "Zweiklang"
[8K2]   11: call ton with "D"
[9RØ]   12: wait
[D41]   13: arg = "3"
[SZ3]   14: ? "Jetzt piepst es nicht"
[WS2]   15: call ton with arg
[3M1]   16: if arg = "X"
[L21]   17:    ? chr(7)
[EC3]   18:    ? "Falsches Argument"
[TUØ]   19: endif
[9RØ]   2Ø: wait
[G12]   21: set talk on
[3VØ]   22: clear
[651]   23: return
```

Listing 3. Ein dBase-Demo-Programm zum Aufruf von »ton.bin«

```
[5P4]    1: code    segment  public 'code'
[PB3]    2:     ;Start des Segments
[VR2]    3: main    proc     far
[3U4]    4: assume cs:code,ds:code,es:code
[Z93]    5: START:  jmp      ANFANG
[P3Ø]    6: ;
[562]    7:     ;Datenbereich
[P3Ø]    8: ;
[617]    9:   AUSGABE:       db      'Dies ist ein Spitze
           nmodul'
[TK1]   1Ø:     ;Länge 25
[114]   11:   EINGABE:       db      25 dup (Ø)
[CP7]   12:     ;Platz für Argument, gleiche Länge beac
           hten!
[HU2]   13:     ;Programmbereich
[P3Ø]   14: ;
[MT2]   15: ANFANG: push    es
[KM3]   16:     ;Sichern der Register
[KB1]   17:   push  ss
[P91]   18:   push  ds
[L91]   19:   push  bx
[P3Ø]   2Ø: ;
[K91]   21:   push  cs
[LV5]   22:     ;es = cs  ds kann erst nach Übernahme
[M71]   23:   pop   es
[KH6]   24:     ;Arguments mit cs gleichgesetzt werden
[KX1]   25:   mov   si,bx
[SG4]   26:     ;Quellstringadresse ds:si
[ØY3]   27:   mov   di,offset EINGABE
[564]   28:     ;Zielstringadresse es:di
[5P1]   29:   mov   cx,25
[2Y2]   3Ø:     ;Länge 25 Zeichen
[ULØ]   31:   cld
[3X1]   32:   rep   movsb
[K91]   33:   push  cs
[ZT5]   34:     ;jetzt kann ds = cs gesetzt werden
[G71]   35:   pop   ds
[P3Ø]   36: ;
[P3Ø]   37: ;
[PT2]   38: SCHLUSS:  pop   di
[ZW1]   39:     ;bx nach di
[M71]   4Ø:   pop   es
[2B6]   41:     ;ds nach es = Zielstringadresse es:di
[U91]   42:   push  es
[PD5]   43:     ;sichern von es (=ds für dBase)
[C14]   44:   mov   si,offset AUSGABE
[C87]   45:     ;Quellstringadresse ds:si  (ds ist noch
          = cs)
[5P1]   46:   mov   cx,25
[ULØ]   47:   cld
[3X1]   48:   rep   movsb
[G71]   49:   pop   ds
[6K4]   5Ø:     ;Zurückholen der Register
[C91]   51:   pop   ss
[M71]   52:   pop   es
[3PØ]   53:   ret
[NB3]   54:     ;Rückkehr zu dBase
[P3Ø]   55: ;
[9K1]   56:   main  endp
[BK1]   57:   code  ends
[CX1]   58:   end   START
```

Listing 4. »wechsel.asm« dient zum Wechseln des empfangenen Arguments

```
[5D5]   1: EB 33 9Ø 44 69 65 73 2Ø 69 73 74 2Ø 65 69 6E 2Ø
[VF5]   2: 53 7Ø 69 74 7A 65 6E 6D 6F 64 75 6C ØØ ØØ ØØ ØØ
[ØZ4]   3: ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
[VB5]   4: ØØ ØØ ØØ ØØ ØØ Ø6 16 1E 53 ØE Ø7 8B F3 BF 1C ØØ
[9R5]   5: B9 19 ØØ FC F3 A4 ØE 1F 5F Ø7 Ø6 BE Ø3 ØØ B9 19
[NW2]   6: ØØ FC F3 A4 1F 17 Ø7 CB
```

Listing 5. Das Hexdump zum Modul »wechsel.bin«

```
[3VØ]    1: clear
[P22]    2: set talk off
[K22]    3: load wechsel
[YY4]    4: arg = "Was ist das für ein Modul"
[2PØ]    5: ? arg
[VE3]    6: call wechsel with arg
[2PØ]    7: ? arg
```

```
[9RØ]    8: wait
[3VØ]    9: clear
[G12]   1Ø: set talk on
[651]   11: return
```

Listing 6. Mit »wechsel.prg« rufen Sie »wechsel.bin« auf

```
[P22]    1: ; schutz.asm
[EN4]    2: ; schaltet Bildschirm an / aus
[UNA]    3: ; wird von dbase III+ mit call with "A" (st
            ring!) = Ausschalten bzw
[PT4]    4: ; "E" = Einschalten aufgerufen
[P3Ø]    5: ;
[VH4]    6: code      segment  public 'code'
[PB3]    7:     ;start des segments
[5R2]    8: main      proc     far
[Z93]    9:     ;dbase verlangt far
[3U4]   1Ø: assume cs:code,ds:code,es:code
[P3Ø]   11: ;
[SW2]   12: ANFANG:   push     es
[KM3]   13:     ;sichern der register
[KG1]   14:   push     ss
[ZF1]   15:   push     bx
[MF1]   16:   push     ds
[P3Ø]   17: ;
[PU2]   18:   mov      ah,ds:[bx]
[ØY2]   19:     ;parameter nach ah
[4V1]   2Ø:   mov      bx,cs
[771]   21:     ;ds = cs
[2V1]   22:   mov      ds,bx
[P3Ø]   23: ;
[NT1]   24:   cmp      ah,'A'
[BN5]   25:     ;in ah befindet sich der buchstabe
[GG1]   26:   je       AUS
[Y14]   27:     ;wenn A dann ausschalten
[XT1]   28:   cmp      ah,'E'
[ØF1]   29:   je       EIN
[514]   3Ø:     ;wenn E dann einschalten
[232]   31:   jmp      FEHLER
[P3Ø]   32: ;
[ZY2]   33: AUS:      call     AUSP
[9Ø4]   34:     ;Aufruf der Prozeduren
[5F2]   35:   jmp      SCHLUSS
[9X2]   36: EIN:      call     EINP
[P3Ø]   37: ;
[PY2]   38: SCHLUSS:  pop      ds
[WØ4]   39:     ;Abschluss des programms
[TD1]   4Ø:   pop      bx
[CE1]   41:   pop      ss
[GD1]   42:   pop      es
[3PØ]   43:   ret
[NB3]   44:     ;Rückkehr zu dbase
[P3Ø]   45: ;
[1T2]   46: FEHLER:   pop      ds
[WW5]   47:     ;Abschluss des programms bei Fehler
[TD1]   48:   pop      bx
[NW1]   49:   mov      ah,'X'
[9W2]   5Ø:   mov      ds:[bx],ah
[CE1]   51:   pop      ss
[GD1]   52:   pop      es
[3PØ]   53:   ret
[NB3]   54:     ;Rückkehr zu dbase
[P3Ø]   55: ;
[G42]   56:     ;prozeduren
[P3Ø]   57: ;
[VZ2]   58: EINP      proc     near
[D52]   59:   mov      ax,ØØ4ØH
[1V1]   6Ø:   mov      ds,ax
[323]   61:   mov      dx,[ds:ØØ63H]
[6D3]   62:     ;liefert Adresse des
[EV1]   63:   add      dx,Ø4H
[BF4]   64:     ;Bildschirmkontrollports
[113]   65:   mov      al,[ds:ØØ65H]
[2S1]   66:   or       al,Ø8H
[PE3]   67:     ;Signal einschalten
[6V1]   68:   out      dx,al
[3PØ]   69:   ret
[1S1]   7Ø:   einp     endp
[P3Ø]   71: ;
[B13]   72: AUSP      proc     near
[D52]   73:   mov      ax,ØØ4ØH
[1V1]   74:   mov      ds,ax
[323]   75:   mov      dx,[ds:ØØ63H]
[EV1]   76:   add      dx,Ø4H
[113]   77:   mov      al,[ds:ØØ65H]
[L32]   78:   and      al,Øf7H
[6G3]   79:     ;Signal ausschalten
[6V1]   8Ø:   out      dx,al
[3PØ]   81:   ret
[HT1]   82:   ausp     endp
[P3Ø]   83: ;
[P3Ø]   84: ;
[9W1]   85: main      endp
[BW1]   86: code      ends
[UØ2]   87:   end      ANFANG
```

Listing 7. Zum gesteuerten Aus- und Einschalten des Bildschirms dient »schutz.asm«

```
[3R5]  1: Ø6 16 53 1E 8A 27 8C CB 8E DB 8Ø FC 41 74 Ø8 8Ø
[1M5]  2: FC 45 74 Ø9 EB ØF 9Ø E8 27 ØØ EB Ø4 9Ø E8 ØE ØØ
[TL5]  3: 1F 5B 17 Ø7 CB 1F 5B B4 58 88 27 17 Ø7 CB B8 4Ø
[DL5]  4: ØØ 8E D8 8B 16 63 ØØ 83 C2 Ø4 AØ 65 ØØ ØC Ø8 EE
[MD5]  5: C3 B8 4Ø ØØ 8E D8 8B 16 63 ØØ 83 C2 Ø4 AØ 65 ØØ
[2E1]  6: 24 F7 EE C3
```

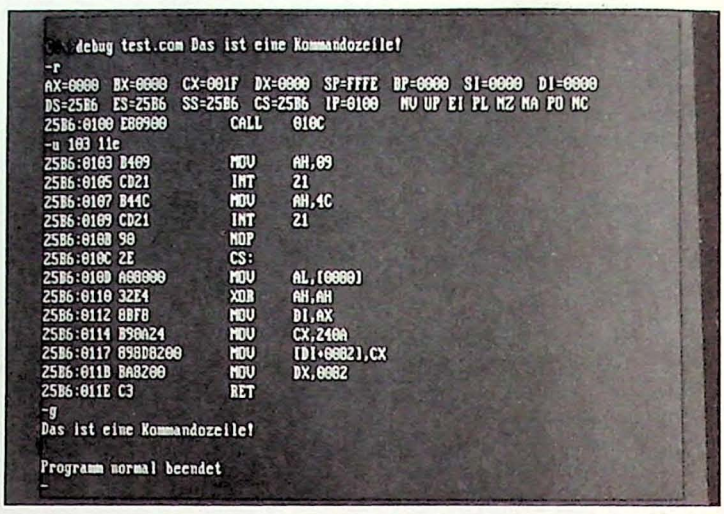
Listing 8. Der Bildschirmschutz »schutz.bin« als Hexadezimal-Listing

```
[TE1]    1: *schutz*
[5W4]    2: *schaltet Bildschirm aus/ein*
[P22]    3: set talk off
[3VØ]    4: clear
[832]    5: load schutz
[HG9]    6: *Hier wird zur Demonstration eines Bildschi
            rminhalts ein
[5H4]    7: *inverses Rechteck erzeugt
[RN2]    8: set color to N/W
[RX2]    9: @ 1Ø,1Ø clear to 2Ø,2Ø
[9P2]   1Ø: set color to W/N
[PD8]   11: ? "Das Programm wartet auf die Eingabe eine
            s Zeichens"
[SU9]   12: ? "Nach der eingestellten Zeit schaltet es
            den Bildschirm ab"
[LY6]   13: ? "und nach Drücken einer Taste wieder an"
[XM8]   14: ? "Eingabe von X (Grossbuchstabe!) beendet
            das Programm"
[FT9]   15: *Als Argument wird zur Abwechslung eine Var
            iable verwendet
[8Ø1]   16: PE = "E"
[CZØ]   17: PA = "A"
[MC1]   18: ZEIT = 2ØØ
[KN8]   19: **2ØØ entspricht bei dBase ca 2Ø sec, bei d
            bFast ca 4 sec**
[MV1]   2Ø: do while .t.
[HNA]   21: *Hauptschleife wird erst verlassen, wenn ei
            n gültiges Zeichen
[AB8]   22: *eingegeben wurde (in unserem Fall ein groß
            es X)
[121]   23:    DEL = Ø
[341]   24:    KEY = Ø
[1B2]   25:    do while KEY = Ø
[1E6]   26:    *Diese Schleife wartet auf eine Eingabe
[SR9]   27:    *solange keine Taste betätigt wird, wird
             der Zähler DEL
[NM9]   28:    *inkrementiert. Beim Erreichen des Endwe
            rts ZEIT wird mit
[4K9]   29:    *schutz.bin der Bildschirm ausgeschaltet
            . Damit sich das
[ZX8]   3Ø:    *nicht ständig wiederholt, wird DEL noch
             um 1 und dann
[KE4]   31:    *nicht mehr weiter erhöht
[S32]   32:       KEY = inkey()
[761]   33:       do case
[9E2]   34:       case DEL < ZEIT
[WR1]   35:          DEL = DEL + 1
[CE2]   36:       case DEL = ZEIT
[WR1]   37:          DEL = DEL + 1
[NE3]   38:          call schutz with PA
[PA1]   39:       endcase
[6VØ]   4Ø:    enddo
[YM8]   41:    *Die Schleife ist verlassen worden, dh.e
            s wurde eine
[RM8]   42:    *Taste betätigt. Ist DEL > ZEIT ist der
            Bildschirm
[RR8]   43:    *abgeschaltet, dh. er wird jetzt wieder
            eingeschaltet
[7X1]   44:    if DEL > ZEIT
[5F3]   45:       call schutz with PE
[TUØ]   46:    endif
[WB7]   47:    *Hier wird auf zulässiges Zeichen geprüf
            t
[WN7]   48:    *Sind mehrere Zeichen möglich, würde hie
            r eine
[A54]   49:    *do case Anweisung stehen
[9K1]   5Ø:    if KEY = 88
[ASØ]   51:       exit
[TUØ]   52:    endif
[6VØ]   53: enddo
[1SØ]   54: ? KEY
[HM4]   55: *Fortsetzung des Programms
[G12]   56: set talk on
[651]   57: return
```

Listing 9. Mit »schutz.prg« aktivieren Sie den Bildschirmschutz

Auf Fehlersuche mit dem Debugger

Mit dem MS-DOS-Debugger »Debug« erhalten die Programmierer eine preiswerte Hilfestellung bei der Fehlersuche in Programmen, bei der Veränderung von Dateien auf der Diskette und der Manipulation von Speicherstellen und Portadressen im PC.



Der MS-DOS-Debugger während der Arbeit

Sicherlich haben sie den MS-DOS-Debugger schon einmal gestartet, seine Befehle ausprobiert und schließlich festgestellt, daß die kurze Anleitung im MS-DOS-Handbuch noch keinen Eindruck von den Fähigkeiten des Debuggers vermittelt: Die vielen Anwendungsgebiete entdeckt man erst nach einiger Einarbeitungszeit.
Gestartet wird der Debugger durch eine Kommandozeile, die bereits Informationen über das zu behandelnde Programm enthalten kann. So bewirkt die Angabe eines Dateinamens, daß »Debug« diese Datei nach dem Start in den Speicher lädt und den Programmzeiger auf den ersten Befehl des Programms setzt.
Manche zu testenden Programme erfordern jedoch die Übergabe von Parametern in der Kommandozeile (so zum Beispiel fast alle MS-DOS-Befehle). Um einen solchen Aufruf mit dem Debugger zu simulieren, werden die Kommandozeilenargumente hinter dem Programmnamen wie üblich eingefügt:

```
debug myprog.com Parameter1 Parameter2
```

Soweit sich der Debugger und das Programm fehlerfrei laden lassen, meldet sich der Debugger mit einem spartanischen Strich. Hat man die gewünschte Datei nicht schon in der Kommandozeile angegeben, so ist dies nun an der Zeit. Dazu ist die Kenntnis zweier Debug-Befehle vonnöten. Debug-Befehle haben zwar einen langen Namen, es darf jedoch jeweils nur der erste Buchstabe eingegeben werden (Tabelle 1). Dieses Zeichen muß als erstes in der Zeile stehen. Hinter dem Befehl werden üblicherweise keine Leerzeichen eingegeben.
Als erstes ist die Festlegung des Namens der zu ladenden Datei notwendig, eventuell auch eine zugehörige Kommandozeile. Diese Eingaben nehmen Sie mit Hilfe des Befehls »name« vor. Dieser Befehl teilt dem Debugger den Namen des Programms mit und erlaubt auch das Simulieren einer Kommandozeile. Um beim vorigen Beispiel zu bleiben: hier würde die folgende Eingabe erwartet:

```
nmyprog.com Parameter1 Parameter2
```

Sollte sich »myprog.com« nicht im aktuellen Verzeichnis befinden, ist eine Pfadangabe erforderlich, zum Beispiel:

```
nc:\assemble\myprog.com Parameter1
Parameter2
```

Nach dem Ausführen dieses Befehls ist die Datei jedoch noch nicht geladen, dies wird vielmehr durch »load« erledigt. Ohne weitere Argumente weist dieser Befehl den Debugger an, die Datei vollständig in den Speicher zu laden. Exe-Dateien werden ab dem Offset 0 eines neuen Segments geladen, alle anderen ab dem Offset 100hex einer Com-Datei. Die Zahl der von der Diskette gelesenen Byte wird in den Registern BX:CX als Doppelwort im Intel-Format gespeichert, also in BX das höherwertige und in CX das niederwertige Wort.

Bevor sich eine Datei laden läßt, muß deren Bezeichnung mit dem Name-Befehl festgelegt werden

Der Load-Befehl kann jedoch auch dazu verwendet werden, um direkt einzelne oder mehrere Sektoren von der Festplatte oder Diskette zu laden. Dadurch kann Debug auf der Festplatte oder Diskette Änderungen vornehmen, die mit MS-DOS-üblichen Mitteln nicht möglich wären. Damit sind natürlich auch gewisse Gefahren verbunden – schlimmstenfalls könnten sämtliche Daten auf der Platte verlorengehen. Solange Sie jedoch nur laden und nicht speichern, kann nichts geschehen.

Weil MS-DOS-Laufwerke logisch nicht in Spuren und Sektoren, sondern lediglich in Sektoren aufgeteilt sind, ist zur Kennzeichnung des Sektors nur eine Zahl notwendig. Der Load-Befehl hat folgende Befehlssyntax:

```
l<Adresse> <Laufwerk> <Sektor>
<Länge>
```

Nachdem alle Parameter hexadezimal angegeben werden, muß für das Laufwerk nicht ein Buchstabe, sondern eine Zahl eingegeben werden. Das Laufwerk »A« entspricht hier einer 0, »B« einer 1 und so weiter. Die Angabe »Sektor« spezifiziert den ersten zu ladenden Sektor, es werden »Länge« Sektoren in den Speicher ab »Adresse« geladen.
Um sich die Daten anzusehen, werden die bekannten Befehle »Unassemble« und »Display« verwendet. Nachdem man dem Prompt (also dem Minuszeichen am Anfang der Zeile) nur wenig Information abringen kann, kommt ein anderer wichtiger Befehl zum Tragen: »Register«. Ohne weitere Parameter eingegeben, zeigt er die Belegung aller Prozessorregister an, und ebenso den Stand der Statusbits (Flags). Die dabei verwendeten Abkürzungen sind in Tabelle 2 angegeben. In der letzten der ausgegebenen Zeilen wird der nächste abzuarbeitende Assemblerbefehl angezeigt. An diese Art der Darstellung müssen Sie sich gewöhnen, sie wird auch beim schrittweisen Abarbeiten des Programms (Trace-Modus) verwendet.
Auf diese Weise gehen Sie nämlich auf Fehlersuche: Befehl für Befehl des fehlerhaften Programms wird angezeigt, abgearbeitet und auf seine Resultate hin überprüft. Die verwendeten Befehle lauten »go«, »trace«, »proc« und steuern den Ablauf des Programms. »g« ohne weitere Parameter startet das Programm an der aktuellen Anweisung und läßt es bis zum Ende durchlaufen – dies wird

jedoch selten gewünscht. Häufig wird man daher den Go-Befehl mit sogenannten Unterbrechungsmarken verwenden. Diese veranlassen den Debugger, das Programm bis zu einer der vorbestimmten Adressen laufen zu lassen:

```
g=0100 011F 0130
```

Mit diesem Befehl wird das Programm an der Adresse 0100hex gestartet. Diese Angabe ist optional und kann auch weggelassen werden, um das Programm an der aktuellen Adresse fortzusetzen. Debug meldet sich erst dann zurück, wenn das Programm entweder bei 011Fhex oder bei 0130hex angelangt ist. Die Angabe mehrerer Unterbrechungen ist sinnvoll, wenn der Programmablauf aufgrund von Verzweigungen im Programm nicht genau vorhersehbar ist. Um einzelne Befehle oder Programmschritte auszuführen, stehen die Befehle »p« und »t« zur Verfügung. Sie gleichen sich in der Syntax:

```
p=0100 5
```

Dieser Befehl weist den Debugger an, ab Adresse 0100hex fünf Programmschritte auszuführen. Beide Parameter sind optional, ohne Startadresse beginnt der Debugger wie gewohnt an der aktuellen Adresse, und ohne Schrittzahl wird ein einziger Schritt ausgeführt. Die Unterschiede zwischen »t« und »p« finden sich in der Auslegung eines »Schritts«: Während »t« immer exakt einen Befehl ausführt, führt »p« Unterprogramme und Interrupts ohne Unterbrechung aus. Dies ist auch sinnvoll: Befindet sich beispielsweise ein MS-DOS-Aufruf im Programm, so können Sie ihn mit »p« ausführen, ohne ihn Schritt für Schritt durchgehen zu müssen (Bild). Das gleiche gilt auch für Unterprogramme – wenn Sie wissen, daß sie fehlerfrei sind, können Sie sie mit »p« rasch ausführen. Nützlich ist »p« auch bei String- und Schleifenbefehlen, die dann ohne Unterbrechung durchgeführt werden. Eine Adresse wird bei den Einzelschrittbefehlen meistens dann angegeben, wenn man einen Befehl überspringen möchte. Haben Sie dann einen Fehler gefunden, ist es rationell, ihn gleich an Ort und Stelle zu beseitigen, um nach weiteren Ausschau zu halten. Dazu dienen die Befehle »enter« zur Eingabe von Daten und »assemble« zum Eingeben von Assemblercode. Erwähnenswert sind einige Eigenheiten der Assemblersprache des Debuggers. Da ist einmal die vorgezogene Eingabe eines Segmentpräfixes: Statt

```
mov ax,cs:[di]
```

ist in Debugger folgendes zu schreiben:

```
cs:
mov ax,[di]
```

Wichtig zu wissen ist auch, wie man dem Debugger die Größe eines unbestimmten Operanden klarmacht, ihm also mitteilt, ob sich der Befehl auf ein Byte oder ein Wort bezieht. Eine solche Situation tritt zum Beispiel beim Inc-Befehl auf:

```
inc byte [0123]
```

signalisiert dem Debugger, daß sich der Befehl auf das an Adresse DS:0123hex befindliche Byte bezieht. Das Gegenstück zu »byte« ist natürlich »word«. Bei den operandenlosen Stringbefehlen des Prozessors wird die Größe durch das direkte Anhängen eines »b« oder »w« an den Befehl angezeigt: »movsb«, »cmpsw« oder »lodsb«. Ansonsten sind bei der Assemblierung mit dem Debugger keine Besonderheiten zu beachten.

Wenn Sie Ihre Änderungen vorgenommen haben, möchten Sie das Programm vielleicht wieder speichern. Der dazu erforderliche Befehl heißt »write«. Ohne Parameter eingegeben, speichert er ab Adresse 0100hex BX:CX Byte unter dem aktuellen Dateinamen ab.

Vorsicht ist geboten, wenn Sie Ihr Programm nach der Verwendung von »go« oder »trace« speichern möchten, weil dann die Programmlänge in den Registern BX und CX nicht mehr enthalten ist. Sie sollten sie daher nach dem Laden des Programms notieren, um sie nun mit dem R-Befehl wieder einzutragen.

Hat das Programm noch keinen Namen, so muß dieser mit dem Befehl »n« eingetragen werden. Sind all diese Voraussetzungen erfüllt, können Sie das Programm speichern:

```
w 0100
```

Exe-Dateien können vom Debugger nicht wieder gespeichert werden.

Natürlich kann der Write-Befehl analog zum Load-Befehl auch explizite Sektoren auf die Festplatte schreiben. Hier ist jedoch äußerste Vorsicht angebracht, da man auf diese Weise auch die FAT (»File Allocation Table«) oder das Inhaltsverzeichnis des Massenspeichers überschreiben kann. Mit dem Befehl

```
w 0100 0 0 10
```

schreiben Sie zum Beispiel ab Speicheradresse 0100hex zehn Sektoren (512 Byte pro Sektor) ab Sektor null auf Laufwerk A. Hüten Sie sich aber davor, zu glauben, daß auf diese Weise eine Datei angelegt wird! Es werden vielmehr die Daten aus dem Speicher direkt auf die Platte geschrieben – ohne Verzeichniseintrag und Aktualisierung der FAT, so daß Sie durch diese Vorgehensweise den Inhalt Ihrer Diskette oder Festplatte zerstören können.

(sw)

Befehl	Syntax	Funktion
Assemble	**a** <Adresse>	Befehle assemblieren
Compare	**c** <Bereich>, <Adresse>	Bereiche vergleichen
Dump	**d** <Bereich>	Speicherinhalt als Hexdump anzeigen
Enter	**e** <Adresse>,<Liste>	Daten eingeben
Fill	**f** <Bereich>,<Liste>	Bereich füllen
Go	**g** = <Adresse>, <Liste>	Programm starten
Hex	**h** <Wert>,<Wert>	Hexadezimalzahlen berechnen
Input	**i** <Portadresse>	Portadresse abfragen
Load	**l** <Adresse><Laufwerk><Sektor><Länge>	Datei oder Sektor laden
Move	**m** <Bereich>, <Adresse>	Bereich verschieben
Name	**n** <Dateiname><Kommandozeile>	Dateinamen festlegen
Output	**o** <Portadresse><Wert>	Portadresse beschreiben
Procstep	**p** = <Adresse><Anzahl>	Programmschritt ausführen
Quit	**q**	Debugger beenden
Register	**r** <Registername>	Registerbelegung anzeigen oder ändern
Search	**s** <Bereich><Liste>	Bytes suchen
Trace	**t** = <Adresse><Anzahl>	Programmbefehl ausführen
Unassemble	**u** <Bereich>	Befehle disassemblieren
Write	**w** <Adresse><Laufwerk><Sektor><Länge>	Datei oder Sektor schreiben

Tabelle 1. Die Syntax aller Debug-Befehle auf einen Blick

Flag	Gesetzt	Gelöscht
Überlauf	OV	NV
Richtung	DN	UP
Interrupt	EI	DI
Vorzeichen	NG	PL
Null	ZR	NZ
Hilfsübertrag	AC	NA
Parität	PE	PO
Übertrag	CY	NC

Tabelle 2. Die Bedeutungen der Abkürzungen bei der Anzeige der Prozessorflags

WERK STATT

Die MASM-5.0-Werkstatt

Die Zeiten sind vorbei, in denen Assemblerprogrammierung sich wirklich nur auf Prozessor und Systemanforderungen beschränkte. Leistungsfähige Assembler wie der MASM 5.0 bieten alle erdenklichen Funktionen, um dem Programmierer bei der Lösung seiner Probleme möglichst tatkräftig beizustehen.

Ein Großteil der MASM-Befehle dient der Organisation von Segmenten, die der Assembler in Klassen und Gruppen zusammenfassen kann. Die meisten Programme können auf diesen Luxus verzichten oder nutzen ihn nur soweit, wie es der Übersetzer fordert. Geht es allerdings darum, Assemblerprogramme mit Programmen, die in anderen Sprachen – vor allem Hochsprachen – geschrieben sind, zu verbinden, bekommen die umständlich anmutenden Segmentvereinbarungen Sinn. Ein Teil von ihnen läßt sich aber auch für die herkömmliche Assemblerprogrammierung einsetzen.

Der MASM 5.0 unterstützt erstmals Speichermodelle, die Microsoft als Standard bei anderen Sprachen eingeführt hat. Mit der Direktive ».model <Modell>« wird möglichst zu Anfang des Programms diese vereinfachte Segmentorganisation aktiviert. Die Bezeichnungen der Modelle und ihre Speicherverwaltung entnehmen Sie Tabelle 1.

Die besondere Bezeichnung »tiny« betrifft Programme, die üblicherweise nur ein Segment groß sind, Programmcode und Daten in diesem einen Segment unterbringen und mit dem Hilfsprogramm »exe2bin« nach dem Linken zu Com-Dateien gewandelt werden. Alle anderen Modelle verwenden schon vorbenannte Segmentbezeichnungen, deren Übereinstimmung mit den Definitionen eines anderen Programmoduls (zum Beispiel eines Moduls einer anderen Sprache) wichtig ist. Ansonsten kann der Linker nicht wissen, welche Segmente zusammengehören. Je nach Modell stehen Direktiven bereit, die entsprechende Segmente vereinbaren. Jede schließt ein vorangegangenes Segment ab, so daß das Kommando » ends« nicht mehr gebraucht wird. Das letzte Segment endet mit der gewöhnlichen End-Klausel. Listing 1 zeigt, wie der Segmentkopf eines Listings unter der vereinfachten Segmentdefinition aussehen kann.

Nun heißen so vereinbarte Segmente aber nicht mehr »code« oder »data« und so weiter, sondern erhalten jene Namen, die alle Microsoft-Compiler ebenfalls verwenden. Ein Programmsegment heißt im Modell »small« dann »__text« und sein Datensegment »__data«.

Außerdem unterscheidet man Datensegmente nach ihrer Belegung. Nicht initialisierte Daten, also solche, die mit »?« und dem Hilfsoperator »dup« zugewiesen wurden, sollten nach der Segmentdirektive ».data?« erwähnt sein. Diese meist großen Felder belegen dann keinen Platz im initialisierten Datensegment. Das erzeugte Segment heißt dann »__bss«. Ebenso sind Konstanten, also Felder, die ihren Wert zur Laufzeit normalerweise nicht ändern, im Segment »const« (Segmentvereinbarung ».const«) besser aufgehoben. Das Segment »stack« behält seinen Namen, es wird mit dem Kommando ».stack« vereinbart.

Alle Microsoft-Sprachen verwenden für ihre Speichermodelle gleiche Segmentbezeichnungen

Ab dem Modell »compact« kommen Sie mit einer weiteren Segmentart in Berührung, mit ».fardata«. Dies ist ein Datensegment, auf das mit Far-Adressen zugegriffen wird, es heißt »far__data«. Das nicht initialisierte ferne Datensegment heißt »far__bss« und ist entsprechend nach ».fardata?« vereinbart.

Welches Speichermodell welche Segmente bereitstellt, sehen Sie in Tabelle 2. Dort erfahren Sie auch, welchen Klassen und Gruppen die vorbenannten Segmente angehören (dazu später mehr). Ab Modell »compact« steht dem Codesegment »__text« der Dateiname voran. Das Codesegment eines Programms mit Namen »test.asm« heißt dann »test__text«.

Wenn Sie im Assemblerquellprogramm verschiedene Segmentnamen verwenden, besteht die Gefahr, hoffnungslos durcheinanderzugeraten. Um dies zu vermeiden, werden vordefinierte Platzhalter verwendet (Tabelle 3). Es empfiehlt sich also, auf die Nennung verschiedener Segmente zu verzichten und mit diesem »Service« zu arbeiten. Müssen Sie aus Platzgründen das Speichermodell einmal wechseln, würden sich die Namen unter Umständen ändern (zum Beispiel von »medium« auf »compact«), und Sie wären dann gezwungen, die neuen Bezeichnungen nachzutragen.

Tabelle 3 zeigt die vorbelegten Konstanten und deren Werte. Diese können Sie auch verwenden, um mit Hilfe bedingter Assemblierung den Quelltext dem Speichermodell anzupassen. Das Segment »stack«, »const« und »data?« ist so nicht in Erfahrung zu bringen. »stack« und »const« sind aber sehr leicht zu merken, und das Segment »data?« ist ohnehin dem Segment »data« angegliedert.

Wer läßt sich aber schon in seinen programmtechnischen Freiheiten einschränken, wenn es nicht unbedingt sein muß? Verfassen Sie eigenständige Assemblerprogramme, dürfen Sie sich Ihre Segmentnamen, -klassen und -gruppen selbst aussuchen. Die generelle Definitionssyntax hierzu lautet

```
<Name> segment <Abstand>
<Kombination> <Adressierung>
'<Klasse>'
```

Modell	Codegröße in kByte	Datengröße in KByte	Kompatibilität zu anderen Microsoft-Sprachen
Tiny	64	muß mit im Codesegment Platz finden	
Small	64	64	MS-C
Medium	>64	64	zu den meisten
Compact	64	>164 (einzelne Datenfelder dürfen jeweils nur 64 KByte umfassen)	MS-C
Large	>64	siehe (»Compact«)	Alle
Huge	>64	>64	zu den jüngeren Versionen

Tabelle 1. Die Segmentorganisation der Speichermodelle ist für alle Microsoft-Sprachen gleich

Modell	Direktive	Bezeichner	Abstand	Verbindung	Klasse	Gruppe
Small	.code	__text	word	public	'code'	
	.data	__data	word	public	'data'	dgroup
	.const	const	word	public	'const'	dgroup
	.data?	__bss	word	public	'bss'	dgroup
	.stack	stack	para	stack	'stack'	dgroup
Medium	.code	<name>__text	word	public	'code'	
	.data	__data	word	public	'data'	dgroup
	.const	const	word	public	'const'	dgroup
	.data?	__bss	word	public	'bss'	dgroup
	.stack	stack	para	stack	'stack'	dgroup
Compact	.code	__text	word	public	'code'	
	.fardata	far__data	para	private	'far__data'	
	.far__data?	far__bss	para	private	'far__bss'	
	.data	__data	word	public	'data'	dgroup
	.const	const	word	public	'const'	dgroup
	.data?	__bss	word	public	'bss'	dgroup
	.stack	stack	para	stack	'stack'	dgroup
Large/Huge	.code	<name>__text	word	public	'code'	
	.fardata	far__data	para	private	'far__data'	
	.far-data?	far__bss	para	private	'far__bss'	
	.data	__data	word	public	'data'	dgroup
	.const	const	word	public	'const'	dgroup
	.data?	__bss	word	public	'bss'	dgroup
	.stack	stack	para	stack	'stack'	dgroup

Tabelle 2. Die vereinfachte Segmentorganisation stellt für jedes Speichermodell verschiedene Segmentbezeichnungen zur Verfügung

Der Segmentname dient der üblichen Zuordnung. Er wird verwendet, wenn das Segment abgeschlossen ist (»<Name> ends«) oder später Register mit Segmentadressen zu laden sind (»mov ax,<Name>). Der Abstand gibt an, an welche Position eines vorangegangenen Segments dieses Segment angeschlossen werden soll. Die sogenannten Schlüsselwörter und was Sie mit den einzelnen Schlüsselwörtern vereinbaren, finden Sie in Tabelle 4.

Mit dem Schlüsselwort »at« setzen Sie Programmsegmente auf absolute Segmentadressen im Speicher

Beginnen Sie ein Segment beispielsweise mit dem Satz

```
data segment word public
```

bestimmen Sie damit den Anschluß dieses Segments an ein bereits mitgebundenes Segment gleichen Namens aus einem vorangegangenen Modul, das bei der nächsten Wortadresse beginnt. Das Segment schließt standardmäßig im Abstand von 16 Byte (»para«) an, wenn Sie den Abstandsschlüssel nicht angeben.
Soll das Segment absolut alleine stehen, ist es vom Kombinationstyp »private«, also nicht verbindbar. Stapelsegmente werden gewöhnlich mit der Kombination »stack« zusammengeschlossen, denn der Stackpointer muß zu Anfang auf das letzte Element im Stack-Segment zeigen. Relativ dazu erfolgt dann die Verbindung.

Soll ein Segment an einer absoluten Adresse beginnen, lautet die Kombination »at <Segmentadresse>«.
Diesen Weg wählen Sie, wenn das Programm auf mehrere Adressen möglicherweise im niederen Speicherbereich zugreifen muß; um also beispielsweise Interrupt-Zeiger auszulesen und zu ändern (was nicht ganz legal ist) oder um Werte zu erfahren, die das Betriebssystem dort bereithält.

```
null segment at 0
```

Hierbei ist das Segment »null« auf das Segment 0 gelegt. Die Assemblerbefehlsfolge

```
mov ax,null
mov es,ax
```

stellt sicher, daß das Register ES auf dieses Segment zeigt. Die Definition im Segment »null«

```
int0 dd ?
int1 dd ?
```

weist den beiden Marken die aktuellen Sprungadressen der Interrupts 0 und 1 zu.

```
org 413h
Platz dw ?
```

übergibt dem Symbol »Platz« den vom Betriebssystem an dieser Stelle freigehaltenen Arbeitsspeicherplatz.
Wenn Sie Segmente anderer Module redefinieren müssen, wenn also auf deren Inhalt ein Zugriff erforderlich ist, wie es die Definition des Segments bislang aber nicht vorsieht, verwenden Sie für das ursprüngliche und für das überlagernde Segment jeweils den Kombinationstyp »common«. Alle Segmente gleichen Namens (und dieser Klausel) beginnen so an der gleichen Adresse (Listing 2 und 3).
Segmentverbindungen geschehen nicht ganz ungeordnet. Ist am Ende der Segmentdefinition eine Klasse angegeben, wird diese Reihenfolge beachtet. Ist keine Verbindung vereinbart, folgen in der späteren Exe-Datei jeweils Segmente gleicher Klassen aufeinander.
Segmente lassen sich zu Segmentgruppen zusammenfassen, wodurch sie nur im Quelltext als getrennte Segmente auftreten, später jedoch zu einem Segment zusammengefaßt werden. Das zeigt aber nur dann Wirkung, wenn Sie statt des Segmentnamens die Gruppenbezeichnung angeben. Der Beschluß

```
<Name> group <Segment>,
<Segment>,<...>
```

faßt alle genannten Segmente zu einem Segment »<Name>« zusammen. Die vorbenannten Segmente »data«, »data?«, »const« und »stack« aus der Modelanweisung sind beispielsweise in einer Gruppe »dgroup« zusammengefaßt.
Wenn Sie in der Objektdatei auf eine bestimmte Reihenfolge der Segmente Wert legen, dann stehen Ihnen die Direktiven ».seq« und ».alpha« zur Verfügung. Im ersten Fall sind alle Segmente in der Reihenfolge ihrer Nennung, im zweiten Fall alphabetisch sortiert. Spätestens der Linker ordnet dann aber nach Klassen oder, wenn er die Option »/dosseg« erhält, in der Reihenfolge, die Sie in Bild 1 sehen. Dieser Schlüssel läßt sich auch im Quelltext verwenden (»dosseg«), um diese Reihenfolge von Anfang an festzulegen. Dabei sollten Sie mit den vorgegebenen Klassennamen arbeiten oder gleich die vereinfachte Segmentorganisation wählen.

Bezeichner	Inhalt
@curseg	momentanes Segment
@filename	Dateiname
@codesize	0 bei Modell »Small« und »Compact«, 1 bei Modell »Medium«, »Large« und »Huge«
@datasize	0 bei Modell »Small« und »Medium«, 1 bei Modell »Compact« und »Large«, 3 bei Modell »Huge«
@code	Name des Segments der Direktive ».code« (»__text« oder »<name>__text«
@data	Name des Segments der Direktive ».data« (»__data«)
@fardata	Name des Segments der Direktive ».fardata« (»far__data«)
@fardata?	Name des Segments der Direktive ».fardata?« (»far__bss«)

Tabelle 3. Mit Hilfe der vorbelegten Konstanten arbeiten Sie stets mit den korrekten Segmentbezeichnungen

Schlüsselwort	Segmentplazierung
byte	byteweise
word	wortweise (zwei Byte)
dword	doppelwortweise (vier Byte)
para	paragraphenweise (16 Byte)
page	seitenweise (256 Byte)

Tabelle 4. Der Anschlußtyp gibt an, in welchem Abstand einzelne Segmente an andere anschließen

Die vereinfachte und die komplexe Segmentorganisation läßt sich beliebig kombinieren. Wählen Sie aber Ihre Segmentnamen mit Überlegung aus, damit sie nicht mit den Standardsegmenten kollidieren. Sie sollten auch unbedingt die Map-Dateien der Compilate anderer Sprachen beachten, denen Sie Ihre Assemblerroutinen hinzufügen. Nicht alle Sprachen folgen der Gruppenzugehörigkeit der Segmente so exakt, wie in Tabelle 2 erläutert.

Müssen Sie nicht mit anderen Sprachen Daten austauschen, definieren Sie sich eigene Segmente

Quick Basic 4.5 beispielsweise weist das Segment »const«, in dem alle »echten« Konstanten lagern, keiner eigenen Gruppe »const« zu, sondern legt es bei »data« ab. Das Überlagern der Segmente anderer Sprachen mit eigenen Assemblersegmenten verläuft zumeist erfolglos, denn sie sind höchst selten als »common« deklariert, also dafür zugelassen.

Um mit den Segmenten letztlich auch arbeiten zu können, müssen Sie sie den Registern zuordnen. Der Assembler benötigt Informationen darüber, wie Sie die Segmentregister verwenden. Nur dann kann der Assembler beispielsweise auch den Versuch beanstanden, auf bestimmte Segmente nicht typengerecht zuzugreifen.

Arbeiten Sie mit Segmentgruppen, ist die Zuweisung schon zur Errechnung der Offset-Adressen notwendig. Hierfür wird jedesmal, bevor Sie ein Segmentregister neu bewerten, der Befehl

```
assume <Register>:<Segment>,<...>
```

verwendet. Damit erklären Sie die Nutzung eines Segments aus Ihrem Quelltext für eines der Segmentregister CS, DS, ES oder SS. Sie dürfen dabei mehrere Register und Segmentnamen durch Kommata getrennt angeben. Als Segmentnamen können Sie auch vorbelegte Textvariablen (Tabelle 3) verwenden.

Der Operator »seg(<Label>)« ist ebenfalls gültig; mit ihm sprechen Sie den Segmentanteil einer Far-Adresse an. Geben Sie statt eines Segmentnamens den Schlüssel »nothing« an, werden damit jegliche Segmentzuweisungen für das Register gelöscht oder aber mit

```
assume nothing
```

alle Segmentzuweisungen.

Nach dem Assume-Kommando besitzen die Register Ihres Assemblerprogramms die Segmentwerte allerdings noch nicht. Sie müssen sie im Quelltext richtig zuweisen. Das Register CS ist auf das Segment des Labels gesetzt, das Sie nach der End-Direktive erwähnt haben. Dorthin zeigen gewöhnlich auch DS und ES. Diese Register müssen Sie mit Assemblerbefehlen wie

```
mov ax,data
mov ds,ax
mov es,ax
```

initialisieren. Wenn Sie für ein Segment den Kombinationstyp »stack« verwendet haben, zeigt das Stack-Segment zu Programmbeginn dorthin. Der Stackpointer SP enthält dann die End-Adresse des letzten Symbols in diesem Segment. In anderen Fällen müssen Sie die zwei Register selbst initialisieren.

MASM bietet einen umfangreichen Funktionssatz, um Datenfelder zu definieren und zu verwalten

Zumeist werden Ihre Segmente Daten enthalten, auf die Sie im Programmverlauf zugreifen müssen. Um diese Daten zu spezifizieren und um sie später im Programm ansprechen zu können, wird ein Label im Segment plaziert und daran ein Datenschlüssel angeschlossen. Dieser Schlüssel legt den Typ des Feldes, wie Sie ihn im Programm verwenden, fest (Listing 2 oder 3). Die grundsätzliche Syntax dazu lautet

```
<Label> <Definitions-Typ>
<Wert(e)>
```

Dem Label schließt sich, anders als bei einer Sprungmarke, kein Doppelpunkt an. Der Definitionstyp für einzelne Bytes heißt »db«, wie Sie in Tabelle 5 sehen. Die gleiche Klausel verwenden Sie, um Strings zu definieren. Diese werden dann mit doppelten oder einfachen Anführungszeichen »"« oder »'« umschlossen.

1.0	Alle Segmente der Klasse »code«
2.0	Segmente, die weder in der Klasse »code« noch in der Gruppe »dgroup« enthalten sind
3.0	Segmente der Gruppe »dgroup«
3.1	Segmente der Klasse »begdata« (reserviert)
3.2	Segmente, die weder der Klasse »begdata« noch »bss« oder »stack« angehören
3.3	Segmente der Klasse »bss«
3.4	Segmente der Klasse »stack«

Bild 1. Die Klausel »dosseg« ordnet die Segmentregister nach diesem Muster

Natürlich muß einer Datendeklaration nicht unbedingt ein Label voranstehen; haben Sie keinen Platz mehr in der Zeile, fahren Sie in der nächsten fort. Konstanten dürfen Sie dabei wie im gesamten Quelltext verwenden. Grundsätzlich geht der MASM von dezimalen Daten aus. Fügen Sie dem Wert ein »h« oder ein »b« an, bestimmen Sie damit die Interpretation der Konstanten als hexadezimale beziehungsweise binäre Zahl. So entspricht der Ausdruck »11« dem Wert 11, »11h« aber dem Wert 17 und »11b« entsprechend der binären Umsetzung 3.

Sie können die Basis, auf der MASM Ihre Konstanten errechnet, auch selbst bestimmen, indem Sie sie mit ».radix <Basis>« vereinbaren. Die Basis »16« bedeutet hexadezimal, »2« binär und »10« wieder dezimal (oder »8« für oktal). Bei der Basis »16« ist allerdings Vorsicht geboten. Endet eine hexadezimale Konstante mit »d« oder »b«, ist sie damit als dezimale beziehungsweise als Binärzahl vereinbart. Im günstigsten Fall resultiert daraus eine Fehlermeldung, zuweilen fügt der MASM aber die falsch errechnete Zahl in die Objektdatei ein. In einem solchen Fall müssen Sie die Konstante explizit als hexadezimal kennzeichnen (Endung »h«).

Soll ein Datenfeld bestimmter Größe einen Wert erhalten, müssen Sie mit

Typ:	byte	word	dword	fword	qword	tbyte
Bytelänge:	1	2	4	6	8	10
Definition:	db	dw	dd	df	dq	dt
Verwendung:	ASCII-Zeichen	Near-Zeiger	Far-Zeiger	lange Integer-Zahlen	Arithmetikprozessor	ungepackte
	niedrige Integer-Zahlen	Integer-Zahlen	hohe Integer-Zahlen	80386-Zeiger		Dezimale
	Flags		Arithmetikprozessorberechnungen			

Tabelle 5. Hier sehen Sie alle Variablentypen, ihren Platzverbrauch und den Definitionsschlüssel

dem Dup-Befehl arbeiten. Nach

```
feld db 10 dup ('*')
```

besitzen zehn Byte ab der Marke »feld« den ASCII-Wert des Sternchens. Wenn Sie statt einer Konstanten ein Fragezeichen angeben (nicht bei Dup-Mengenangaben), reserviert der MASM lediglich das deklarierte Feld, ändert aber dessen Werte nicht. Dies bietet sich für die direkte Adressierung des Bildschirms oder der Zeiger des Null-Segments an. Auch zur Festlegung von Pufferbereichen ist das Fragezeichen beim späteren Lesen von Quelltexten weniger verfänglich.

Fast wie in Pascal lassen sich zusammengehörige Datenfelder zu Strukturen zusammenfassen

Müssen Sie mit Strings fremder Sprachen jonglieren, so hilft Ihnen Bild 2 weiter. Dort ist das Format der Zeichenketten aller wichtigen Programmiersprachen beschrieben. Dies ist jedoch nur dann relevant, wenn Sie Strings lesen oder verändern müssen, deren Adressen Ihnen übermittelt wurden. Mit überlagernden Segmenten kommen Sie so also nicht sehr weit.

Wer zuweilen in Pascal programmiert, dem sind definierbare Datentypen sicher bekannt. Der MASM kennt diese Funktion ebenfalls, nennt sie jedoch »Structures«. In Listing 1 sehen Sie, wie die Typendefinition auszusehen hat. Der Befehl

```
<Name> struc
```

legt eine Datenstruktur unter »<name>« fest. Dem folgen Datendefinitionen durch Befehle wie »db«, »dw« und so weiter (Tabelle 5). Mit der Art der Definition bestimmen Sie zugleich seine Funktion. Die folgende Vereinbarung

```
<Symbol1> db <Wert>
```

legt den ersten Bestandteil der Struktur fest. Er ist ein Byte groß und mit »<Wert>« initialisiert.

```
<Symbol2> db 10 dup (0)
```

Diese Befehlsfolge definiert quasi nur einen Abstand zum nächsten Element, denn sobald Sie unter einem Symbol mehrere Einheiten (Byte bei »db« oder Wörter nach »dw«) festlegen, dürfen Sie sie später nicht mehr definieren. Eine Ausnahme bilden Strings, die zwar einer Folge von mehreren Bytes entsprechen, aber in diesem Fall nur als ein Element betrachtet werden.

```
<Symbol3> db '<Zeichenkette>'
```

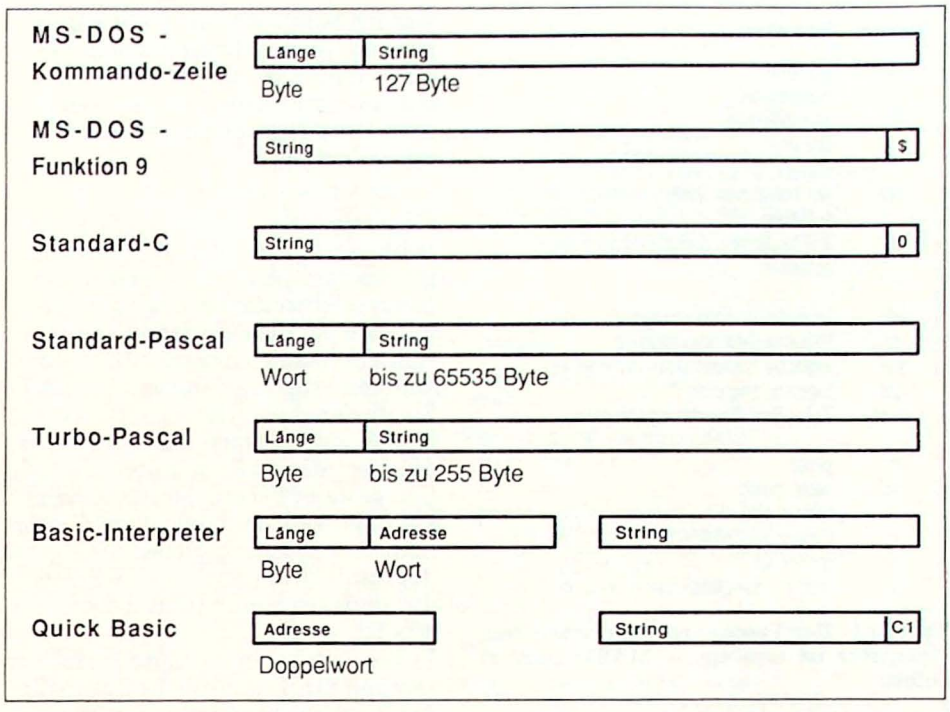


Bild 2. Hier sehen Sie die unterschiedliche String-Verwaltung der wichtigsten Sprachen

Nun noch eine konstante Mehrfachvereinbarung:

```
<Symbol4> db 0dh,0ah,24h
```

und sie schließen die Struktur ab mit

```
<Name> ends
```

Wann immer Sie von nun an eine Datenstruktur dieses Aussehens in einem entsprechenden Datensegment ablegen wollen, geben Sie an:

```
<Feld> <Name> <Wert1,,Wert3,>
```

Die spitzen Klammern um die letzten Elemente sind ausnahmsweise genau so einzugeben. Sonst sind damit jeweils Platzhalter umgeben, für die Sie sinnvolle Angaben einsetzen. Die zwei nebeneinanderstehenden Kommata lassen ein Leerfeld frei, genauso wie das überzählige Komma am Ende des Feldes. Der zweite und vierte Bestandteil der Struktur darf ja nicht mehr umbelegt werden, seine Werte sind konstant. Wenn Sie in dem vereinbarten Datenfeld den Initialisierungswert aus der Strukturdefinition verwenden wollen, gehen Sie ebenso vor, machen für das Feld also keine Angabe.

Auf die Felder eines strukturierten Datenfeldes greifen Sie zu, indem Sie als Marke den Namen des strukturierten Datenfeldes angeben, gefolgt von einem Punkt und dem Namen des Feldelements. Um also eine MS-DOS-Ausgabe (Interrupt 21hex, Funktion 9) vorzubereiten, laden Sie das DX-Register mit der effektiven Adresse des Strings wie folgt:

```
lea dx,<Feld>.<Symbol3>
```

Strukturen kennt der MASM auch auf Bitebene. Bitmuster lassen sich so übersichtlicher verwerten. Wenn Sie ein Zeichen auf dem Bildschirm beispielsweise mit einem passenden Attributbyte versehen wollen, müssen Sie sich in Zukunft nicht mehr mit unleserlichen Aufführungen herumschlagen wie:

```
mov es:[bx],87h
```

Mehr Übersichtlichkeit im Programm mit der Verwendung von Bitsätzen, sogenannten »records«

Viel ästhetischer nimmt sich die Verwendung eines sogenannten »record« aus.

```
Attrib record
blink:1,hint:3,lite:1,vorn:3
```

Dieser Bitsatz unterteilt ein Byte in ein Bit für das Blinkattribut, drei Bit für die Hintergrundfarbe, ein Bit, das für die Leuchtintensität zuständig ist, und schließlich die letzten drei Bit für die Vordergrundfarbe. Daß im oberen Beispiel das Zeichen in normaler Intensität blinkend weiß erscheinen wird, ist dem Hex-Code kaum zu entnehmen. Nach der Definition des Bitsatzes müssen zur Anwendung der Name des Bitsatzes (in jenem Fall »attrib«) und danach in spitzen Klammern die einzelnen Bitwerte angegeben sein.

```
mov es:[bx],attrib <1,0,0,7>
```

Das funktioniert auch mit Wörtern (16 Bit), und mit der Direktive ».386« sind für entsprechende Computer auch 32-Bit-Sätze gestattet. Ähnlich der Struktur-Vereinbarung dürfen die Elemente eines Bitsatzes auch mit

Operator	Bedeutung
+	Addition
-	Subtraktion
*	Multiplikation
\	Division
shl	um bestimmte Stellenzahl nach links schieben
shr	um bestimmte Stellenzahl nach rechts schieben
and	logische Und-Verknüpfung
or	logische Oder-Verknüpfung
xor	logische Exklusiv-Oder-Verknüpfung
not	logische Negation
mod	Teiler-Rest-Berechnung (Modulo)
eq	gleich
ne	nicht gleich
lt	kleiner als
le	kleiner oder gleich
gt	größer als
ge	größer oder gleich

Tabelle 6. Der Taschenrechner neben dem Computer ist unnötig – MASM rechnet selbst

Werten vorbelegt werden, die Sie dann übernehmen, wenn Sie eine Stelle zwischen den Klammern frei lassen. Gehen Sie von einem grundsätzlichen Schwarz-auf-Weiß-Attribut aus, lautet die Satzvereinbarung:

```
Attrib record
blink:1=0,hint:3=0,lite:1=0,vorn:3=7
```

Der vorher ausgeführte Befehl beschränkt sich dann nur noch auf

```
mov es:[bx],attrib <1,,,>
```

Die freigelassenen Stellen bedeuten: Übernahme aus der Vorbelegung. Mit dem zusätzlich eingebrachten Schlüssel »mask« lassen sich ganze Bitsätze oder einzelne Elemente daraus für Boolesche Verknüpfungen im Quelltext benutzen.

```
or es:[bx],mask lite
```

Dieses Beispiel aktiviert das Helligkeitsattribut im zuvor gesetzten Attribut. Schließlich steht noch ein Operator »width« bereit, der die Bitlänge eines Satzes oder eines seiner Elemente übergibt. Das Ergebnis des Ausdrucks »width attrib« lautet dann 8, und das von »width vorn« lautet 3. Der Operator läßt sich für bedingte Assemblierung anwenden.

Der Programmtext beginnt ebenso, wie die Datendefinition endete: mit Symbolen

Nachdem nun sämtliche Daten in Segmenten untergebracht, initialisiert und unterteilt sind, beginnen Sie mit dem Verfassen des eigentlichen Quelltexts. Dabei stoßen Sie wahrscheinlich zu allererst wieder auf Marken. Nur sind es diesmal zur Abwechslung Sprungmarken. Sie werden beispielsweise nicht umhinkönnen, größere Programme in Module zu unterteilen, die sich untereinander mit »call« aufrufen. Der Beginn einer Prozedur wird mit der Vereinbarung

```
<Name> proc <Entfernung>
```

eingeleitet. Für »Entfernung« ist der Schlüssel »near« oder »far« zulässig, je nachdem, ob zu der Prozedur ein naher oder weiter Sprung zu unternehmen ist, die aufrufende Routine sich also im gleichen oder in einem anderen Programmsegment befindet. Sie dürfen diese Klausel auch weglassen, zum Beispiel dann, wenn Sie die vereinfachte Segmentorganisation gewählt haben, der Assembler kümmert sich dann selbst um die korrekte Adressierung. Haben Sie eigene Segmente deklariert, ist eine Prozedur ohne Entfernungsangabe grundsätzlich »near«.

Es kann vorkommen, daß Sie einen zweiten Einsprungspunkt innerhalb der Prozedur benötigen, zum Beispiel, um ein Programmteil zu übergehen. Hierzu legen Sie am Einsprungspunkt mit der Klausel

```
label <name> <Entfernung>
```

die Sprungmarke fest, mit der Sie sich dann im aufrufenden Programmteil beschäftigen. Eine Prozedur endet mit

```
<Name> endp
```

Sämtliche Ret-Befehle in der Prozedur sind gemäß Entfernung »far« oder »near«. Ein Fehler bei der Bezeichnung der Entfernung führt zum überzähligen oder unterlassenen Abziehen einer Adresse vom Stack, so daß mit verwirrenden Ablaufergebnissen zu rechnen sein dürfte. Zumeist wird der Assembler solche Irrtümer aber beanstanden. Sprünge innerhalb des Segments realisieren Sie mit der Benennung der Adresse durch ein Label

```
<Label>: <Assembler-Befehl>
```

und späteren Verzweigungs- oder Sprungbefehlen.

```
jmp <Label>
```

Wenn Sie Prozeduren aus anderen Modulen (Objektdateien) verwenden, müssen Sie vor dem Aufruf bekanntgeben, daß der Einsprung sich nicht im gegenwärtigen Modul befindet.

```
extrn <Label>:<Entfernung>
```

Die Marke muß in ihrem Programm-Modul dann auch als »public« deklariert sein, damit der Linker weiß, welche Labels globaler Natur sind.

```
public <Label>
```

Fehler, die durch unsachgemäße Handhabung externer Marken verursacht werden (zu denen zählen übrigens auch Datenmarken), kommentiert der Linker mit der Meldung »Unresolved Externals«.

Bei der Analyse einsegmentiger Programme, die zu Com-Dateien umgewandelt werden sollen, ist Ihnen sicherlich bereits der Schlüssel »org 100h« am Anfang aufgefallen. Es handelt sich dabei lediglich um eine Vorwärtsverschiebung des Adreßzeigers um 256 Byte, damit das dort gelagerte Programmsegmentpräfix (PSP) von MS-DOS nicht überschrieben wird.

Sie können das Kommando immer dann anwenden, wenn die Verschiebung des Adreßzeigers erforderlich ist, beispielweise um belegte Datenbereiche nicht zu überschreiben. Zuweilen ist es aber auch notwendig, den Adreßzeiger auf die nächste Wortadresse oder auf die nächste Adresse, die ein Vielfaches einer bestimmten Zahl ist, zu setzen. Wenn Sie zum Beispiel mehrere Strings nacheinander definieren, im Programm aber davon ausgehen wollen, daß ihre Anfänge alle den gleichen Abstand voneinander haben, um sie leichter adressieren zu können, verwenden Sie die Align-Direktive. Nach

```
align 10h
```

befindet sich der Adreßzeiger unter Garantie auf dem Anfang des nächsten nicht angebrochenen Paragraphen, Bereiche dazwischen werden nötigenfalls übergangen. Legen Sie an dieser Stelle den String fest, und wiederholen Sie das Align-Kommando. Der zweite String beginnt ebenfalls erst beim nächsten Paragraphen. Durch mehrfaches Schieben nach links (»rol«) bringen Sie dann zum Beispiel ein Index-Register dazu, auf den jeweils nächsten Paragraphen zu zeigen.

Der Befehl »align 2« bewegt den Adreßzeiger zur nächsten geraden Adresse, die Direktive »even« erfüllt den gleichen Zweck. Die vorbelegte Variable »$« enthält übrigens den aktuellen Adreßzeiger.

Damit Ihre Arbeit mit dem Assembler nicht durch zwischenzeitliches Suchen nach Taschenrechner oder Rechenschieber aufgehalten wird, akzeptiert der MASM Rechnungen innerhalb des Quelltextes. Deren Ergebnisse setzt er dann in den Programmcode ein. In Tabelle 6 sehen Sie, welche Rechenarten Ihnen hierfür zur Verfügung stehen.

Nun wissen Sie über den MASM genug, um seine wirklich überragenden Hilfen in Ihrer Programmierung einzusetzen. Themen wie »bedingte Assemblierung« oder »Makroverarbeitung« wurden bewußt nicht aufgenommen, sie werden an anderer Stelle in dieser DOS-Extra-Ausgabe behandelt.

(Dieter Bäumler/sw)

```
[XY1]    1: .model small
[ØØØ]    2:
[ØD2]    3: MSDOS    equ 21h
[PR2]    4: SetDate equ 2bh
[CC2]    5: Exit     equ 4ch
[ØØØ]    6:
[T12]    7: datum struc
[6F1]    8: Tag      db 1
[7N1]    9: Monat    db 1
[112]   1Ø: Jahr     dw 198Ø
[DP1]   11: datum ends
[ØØØ]   12:
[D21]   13: .stack
[2SØ]   14: .data
[S14]   15: @filename  datum <1,6,1989>
[FTØ]   16: .code
[EØ4]   17: assume CS:@code,ds:@data
[3A3]   18: Start:  mov ax,@data
[1H1]   19:         mov ds,ax
[XZ4]   2Ø:         mov dl,byte ptr @filename.Tag
[4G5]   21:         mov dh,byte ptr @filename.Monat
[675]   22:         mov cx,word ptr @filename.Jahr
[TØ3]   23:         mov ah,low SetDate
[4M1]   24:         int MSDOS
[L12]   25:         mov ah, Exit
[4M1]   26:         int MSDOS
[4K1]   27: end Start
```

Listing 1. Programmierung der vereinfachten Segmentorganisation – »today.asm« setzt das Datum auf den eingestellten Tag

```
[CH1]    1: Comment *
[L23]    2: Programm: Rufe.asm
[M24]    3: Autor    : Dieter Bäumler
[PF6]    4: (c)1989 Redaktionsbüro Everts & Hagedorn
[833]    5:         DOS International
[ØM7]    6: Dieses Programm muß assembliert und dann
[YZ6]    7: mit »link rufe+zeige« zusammengebunden
[L75]    8: werden. Es zeigt den Text aus dem
[KA6]    9: geteilten Segment in ASCII-Zeichen und
[DE7]   1Ø: ruft danach die Prozedur »Zeige« im Modul
[BA3]   11: »zeige.asm« auf.
[M2Ø]   12: *
[UNØ]   13: .seq
[MF5]   14: .Radix 16              ; Basis hexa-dezimal
[F33]   15: extrn Zeigen: near
[812]   16: public Start
[YN3]   17: GRUPPE group DATA,FELD
[ØØØ]   18:
[1L4]   19: STACK segment stack 'stack'
[ER1]   2Ø: db 1ØØ dup (?)
[2P1]   21: STACK ends
[ØØØ]   22:
[K94]   23: FELD segment common 'data'
[BE6]   24: Alpha db 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
[ØK1]   25:       db Ødh,Øa,24
[ØG1]   26: FELD ends
[ØØØ]   27:
[L94]   28: DATA segment public 'data'
[VA6]   29: Text  db 'Dieser Programmteil gibt das'
[M43]   3Ø:       db ' Feld ASCII aus.'
[4H2]   31: CRLF  db Ødh,Øa,24
[DG1]   32: DATA ends
[ØØØ]   33:
[CA4]   34: CODE segment public 'code'
[NX5]   35: assume CS:CODE, DS:GRUPPE, ES:GRUPPE
[3P2]   36: assume SS:STACK
[PS6]   37: Start:  mov dx,GRUPPE ;DS initialisieren
[G52]   38:         mov ds,dx     ;
[P75]   39:         lea dx,Text   ;Offset des Textes
[1D4]   4Ø:         mov ah,9      ;für Funktion 9
[PX3]   41:         int 21        ;des MS-DOS-Int
[NZ1]   42:         lea dx,Alpha
[N63]   43:         mov ah,9      ;jeweils
[FU1]   44:         int 21        ;
[UA2]   45:         lea dx,CRLF   ;
[AV3]   46:         mov ah,9      ;desgleichen
[3ZØ]   47:         int 21
[1U4]   48:         call Zeigen   ;Modul aufrufen
[AC4]   49:         mov ah,4c     ;Funktion Exit
[PX3]   5Ø:         int 21        ;des MS-DOS-Int
[TF1]   51: CODE ends
[4K1]   52: end Start
```

Listing 2. »rufe.asm« muß mit »zeige.asm« gelinkt werden; die Programmteile verwenden ein überlagerndes Segment

```
[CH1]    1: Comment *
[ØK4]    2: Programm-Modul: Zeigen.asm
[ME4]    3: Autor         : Dieter Bäumler
[PF6]    4: (c)1989 Redaktionsbüro Everts & Hagedorn
[833]    5:         DOS International
[P96]    6: Dieses Programm-Modul gibt den Text im
[EG6]    7: überlagerten Segment hexa-dezimal aus.
[M2Ø]    8: *
[UNØ]    9: .seq
[HA1]   1Ø: .radix 16
[ØØØ]   11:
[YN3]   12: GRUPPE group DATA,FELD
[A42]   13: public zeigen
[ØØØ]   14:
[1L4]   15: STACK segment stack 'stack'
[2P1]   16: STACK ends
[ØØØ]   17:
[L15]   18: FELD segment para common 'data'
[XX2]   19: daten db 1a dup (?)
[ØG1]   2Ø: FELD ends
[ØØØ]   21:
[Y35]   22: DATA segment byte public 'data'
[736]   23: Text      db 'Dieser Programmteil gibt '
[YV4]   24:           db 'das Feld hexa-dezimal aus.'
[4V2]   25: CRLF      db Ødh,Øa,24
[XA4]   26: Digits    db 'Ø123456789ABCDEF'
[9E2]   27: ThisDigit db Ø,Ø
[SH2]   28: Abstand   db 2Ø,24
[DG1]   29: DATA ends
[ØØØ]   3Ø:
[Ø15]   31: CODE segment para public 'code'
[NX5]   32: assume CS:CODE, DS:GRUPPE, ES:GRUPPE
[3P2]   33: assume SS:STACK
[UT2]   34: Zeigen  proc near
[PC1]   35:         push ax
[6Y2]   36:         push bx          ; alle
[2A4]   37:         push cx          ; verwendeten
[BT3]   38:         push dx          ; Register
[TC3]   39:         push si          ; retten
[W33]   4Ø:         mov dx,GRUPPE    ; DS
[9W4]   41:         mov ds,dx        ; initialisieren
[8L2]   42:         push ds          ; ES
[NN4]   43:         pop es           ; initialisieren
[865]   44:         lea dx,Text      ; Text-Offset für
[FE4]   45:         mov ah,9         ; Funktion 9 des
[PP3]   46:         int 21           ; MS-DOS-Int
[KN4]   47:         lea bx,daten     ; Basis-Zeiger
[L84]   48:         mov si,Ø         ; und Index =  Ø
[X36]   49: schl:   mov al,es: [bx+si] ; Byte holen
[153]   5Ø:         call HexOut        ; ...
[BØ4]   51:         inc si             ; nächster
[334]   52:         cmp si,1ah         ; letzter ?
[4C4]   53:         jne schl           ; nein, weiter
[9L2]   54:         lea dx,CRLF        ;
[484]   55:         mov ah,9         ; CRLF ausgeben
[3ZØ]   56:         int 21
[894]   57:         pop si           ; alle Register
[583]   58:         pop dx           ; wieder
[Z74]   59:         pop cx           ; zurückholen
[T31]   6Ø:         pop bx
[R31]   61:         pop ax
[RH3]   62:         ret              ; nach Hause
[PY1]   63: zeigen  endp
[ØØØ]   64:
[CW2]   65: HexOut  proc near
[ZC1]   66:         push dx
[8F3]   67:         push cx      ; Register
[233]   68:         push bx      ; retten
[PC1]   69:         push ax
[9G4]   7Ø:         mov cl,4     ; Byte 4 mal nach ...
[C33]   71:         xor bx,bx    ; kill bx
[7C3]   72:         mov bl,al    ; Byte -> bx
[PX4]   73:         shr bl,cl    ; ... rechts schieben
[L85]   74:         mov dl,[offset Digits+bx] ; Hex-
[DL5]   75:         mov byte ptr ThisDigit,dl ; Digit
[VP4]   76:         mov bl,al        ; unterer Nibbel
[LW3]   77:         and bl,Øf        ; isolieren
[LY4]   78:         mov dl,[offset Digits+bx]   ; ...
[EB5]   79:         mov byte ptr ThisDigit+1,dl ; ...
[GU4]   8Ø:         lea dx,ThisDigit  ; Zeiger auf
[CS3]   81:         mov ah,9          ; Zahl und
[NL3]   82:         int 21            ; ausgeben
[R31]   83:         pop ax
[GR4]   84:         pop bx    ; Registers come back
[V31]   85:         pop cx
[X31]   86:         pop dx
[6H4]   87:         ret       ; zurück zum Rufer
[812]   88: HexOut  endp
[ZP1]   89: code    ends
[4MØ]   9Ø: end
```

Listing 3. »zeige.asm« ist der Programmteil, der das Datenfeld hexadezimal anzeigt

WERK STATT

Die AFD-Pro-Werkstatt

Der AFD-Pro ist einer der leistungsfähigsten Debugger auf dem Markt. Kaum einer seiner Konkurrenten läßt sich derart einfach bedienen und zeigt so viele Informationen so übersichtlich auf dem Bildschirm. Sein Aufbau gestattet sogar das Einbinden von benutzerdefinierten Befehlen. Erfahren Sie nun, wie Sie Befehlserweiterungen für den AFD-Pro programmieren und diesen dabei für das unvermeidliche Debugging einsetzen können.

Ist ein Programm im Quellcode erfaßt, steht erst die tatsächliche Arbeit an: die Beseitigung der Fehler. Die syntaktischen Fehler sind noch sehr schnell behoben, den logischen Fehlern eines Assemblerprogramms ist allerdings schwerer beizukommen, als das in jeder anderen Sprache der Fall ist. Fehlermeldungen, die auf falsche Werte hinweisen, gibt es nicht, Rechnerabstürze und unsinnige Ergebnisse sind die einzigen Merkmale, die auf ein fehlerhaftes Programm schließen lassen. Auch wenn ein Programm augenscheinlich das tut, was von ihm verlangt wird, ist damit zu rechnen, daß es Fehler hat. Fallunterscheidungen sind unter Assembler in kleinstmögliche Bestandteile aufgeteilt. Unzählige Testläufe wären erforderlich, um sie alle zu durchlaufen. An dieser Stelle tritt der Debugger in Aktion. Mit seiner Hilfe können Sie in jeder Phase des Programms Einblick in die Register nehmen oder sie modifizieren und so den korrekten Ablauf prüfen.

Der Befehlsumfang des AFD-Pro (»Advanced Fullscreen Debugger«) ist so umfangreich, daß nötige Erweiterungen eigentlich nur noch anwendungsspezifisch sein können. Das einzige, was dieser Debugger dem Fehlersuchenden bislang noch nicht bietet, sind ein paar aufmunternde Worte.

Listing 1 (»afdload.asm«) ist ein Assemblerprogramm, das den AFD-Pro befähigt, auf Verzweiflungsausrufe des Anwenders tröstend zu antworten, allerdings hat es Fehler. Mit diesem Programm werden jedoch dennoch zwei Fliegen mit einer Klappe geschlagen: Zum einen können Sie das AFD-Debugging üben, zum anderen eine sinnvolle, anwenderspezifische Erweiterung programmieren.

Der grundsätzliche Aufbau einer Befehlserweiterung für den AFD-Pro ist im Handbuch exakt erklärt. Rufen Sie den Debugger mit der Option

```
afdpro /U<Interrupt-Nummer>
```

auf, löst er den vereinbarten Interrupt aus, sobald er einen eingegebenen Befehl nicht kennt. Üblicherweise würde dann eine Fehlermeldung erscheinen. Der Interrupt soll auf eine Routine verbogen sein (vergleiche Listing 1), die der Befehlszeile ein sinnvolles Kommando zu entnehmen versucht und es ausführt. Diese Routine empfängt einen Zeiger in den Registern DS:SI, der auf den Eingabe-String deutet. Das Register CX weiß, wie lang die Eingabe ist.

Laut AFD-Pro-Handbuch steht in ES:DI der Zeiger auf die Benutzerroutine selbst und in AX die Nummer des ausgelösten Interrupts. Die Routine hat schließlich in AL zu übergeben, ob der Befehl fehlerhaft ist. In diesem (Fehler-) Fall muß ES:DX auf einen Fehlermeldungstext zeigen, dem ein Nullbyte folgt. Außerdem steht in SI diejenige Position im Eingabe-String, von der an der Befehl nicht mehr zu identifizieren war.

Nun ergeben sich zwei Alternativen: Der erste Weg bestünde darin, ein residentes Programm zu entwerfen, das eine solche Routine im Speicher hält. Allerdings bleibt es dann dort, auch wenn die Arbeit mit dem Debugger längst abgeschlossen ist. Die wesentlich elegantere Lösung ist ein Programm, das seinerseits AFD-Pro lädt und mit ihm zusammen endet. Dieses Programm ist dann auch in der Lage, den verbogenen Interrupt wieder »geradezurücken«. Unter Umständen wird er ja von einem anderen Programm noch gebraucht.

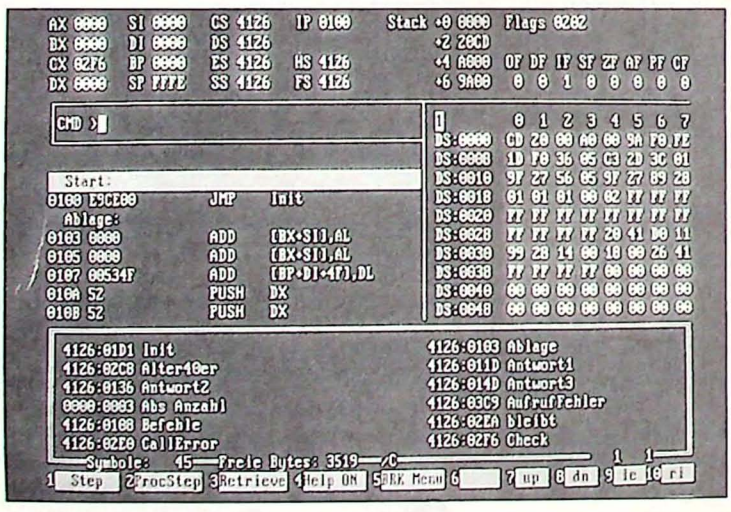


Bild 1. Der Bildaufbau des AFD-Pro ist sehr übersichtlich und garantiert angenehmes Arbeiten

Für die Programmierung eines AFD-Pro-Laders sind verschiedene Kleinigkeiten zu berücksichtigen

Bei dieser Lösung muß man allerdings noch verschiedenes beachten: Der AFD-Befehl »quit r« verläßt zwar den Debugger, doch bleibt er dann resident im Speicher und läßt sich mit der Tastenkombination [Ctrl-Esc] oder einem NMI (»Non-maskable Interrupt«) wieder hervorholen. In diesem Fall muß die Befehlserweiterung also auch resident bleiben. Des weiteren ist die Kommandozeile für den AFD-Pro zu übernehmen, mit der das Ladeprogramm aufgerufen wurde. Darüber hinaus muß das Programm prüfen, ob es nicht schon resident im Speicher ist, bevor es beginnt, Vektoren auf sich zu biegen oder sich seinerseits resident zu installieren.

Assemblieren Sie Listing 1 mit dem MASM oder TASM und der Option »/l«, damit eine Listing-Datei erzeugt wird, die für das Debugging Voraussetzung ist:

```
tasm /l afdload.asm
```

Danach linken Sie die Objektdatei »link afdload /map;« und wandeln diese mit »exe2bin« in eine Com-Datei, oder Sie verwenden

```
tlink /t /m afdload
```

mit den angegebenen Optionen. Erzeugen Sie nun eine Batch-Datei mit

diesen Arbeitsgängen, denn im Laufe der Werkstatt werden diese noch häufiger vonnöten sein. Wenn Sie auf den ästhetischen Genuß der wechselnden Groß- und Kleinschreibung bestehen, sollten sie mit dem zusätzlichen Schalter »/ml« assemblieren und mit »/noignorecase« linken, beziehungsweise »/c«, wenn Sie mit dem »Tlink« des TASM arbeiten. Das erhöht die Übersichtlichkeit beim Umgang mit dem Debugger, zwingt Ihnen aber auf, dabei alle Labels in der richtigen Schreibweise anzugeben.

Wie bereits erwähnt, ist das Programm so nicht lauffähig. Um es zu verbessern, folgen Sie dieser Werkstatt, wobei Sie beachten müssen, daß alle Angaben von Zeilennummern und Adressen sich auf den Quelltext beziehungsweise das Programm im Stadium fortschreitender Verbesserung beziehen. Zumeist sind Einfügungen vorzunehmen, welche die Länge des Textes beinflussen. Möchten Sie die Befehlserweiterung lediglich anwenden, so geben Sie den Hexdump in Listing 2 oder das Turbo-Pascal-Programm in Listing 3 ein, wenn Sie über den dafür notwendigen Compiler verfügen. Unter Turbo Pascal 4.0 benötigen Sie außerdem Listing 4, um es an Zeile 66 in Listing 3 mit der Blockfunktion [Ctrl-K][Ctrl-R] einzufügen. Es handelt sich um die Funktion »GetEnv«, die in der Turbo-Pascal-Version 5.0 bereits enthalten ist.

Für die Verbesserungsarbeiten am Assemblercode muß sich der AFD-Pro von dem Verzeichnis aus, auf dem Sie arbeiten, aufrufen lassen. Entweder muß er sich in demselben Verzeichnis befinden oder in einem in der Umgebungsvariablen »path« aufgeführten Verzeichnis.

»afdload« lädt den Debugger ohne Probleme, steigt bei Eingabe eines unbekannten Befehls aber aus

Wenn Sie »afdload« nun starten – immerhin lädt das Programm bislang anstandslos den Debugger – wird zunächst ein »L« in der Kommandozeile präsentiert und die Bemerkung, daß der »Dateiname fehlt«. Außerdem führt jede Eingabe, die kein AFD-Befehl ist, zum sofortigen Ausstieg, was einen Boot-Vorgang nötig macht. Ist Ihr Rechner wieder betriebsbereit, starten Sie den Debugger mit:

```
afdpro afdload.com
```

Darauf lesen Sie die Symbole aus der Map-Datei:

```
ls afdload.map
```

Sie bekommen eine Fehlermeldung, die aussagt, daß die Datei nicht das erforderliche Format hat (»Ungültige .MAP-Datei«). Verlassen Sie den Debugger (»quit«), und lassen Sie sich die Map-Datei auf dem Bildschirm ausgeben:

```
type afdload.map
```

Sie werden erkennen, daß lediglich das Codesegmentsymbol erwähnt ist. In der Map-Datei tauchen üblicherweise nur Symbole auf, die als »public« definiert sind. AFD-Pro ist aber in der Lage, die Symbole aus einer Lst-Datei zu lesen, die vom Assembler erzeugt wurde. Dazu muß aber mindestens ein Label als »public« deklariert und somit in der Map-Datei erwähnt sein. Starten Sie also Ihren Editor und laden Sie »afdload.asm«. Fügen Sie als Zeile 6 die Anweisung

```
public Init
```

ein. Sofern Sie mit dem TASM assemblieren, müssen Sie dafür sorgen, daß alle Labels in der Map-Datei vorkommen, denn AFD-Pro kann die vom TASM angelegte Lst-Datei nicht nachladen. Listing 6 enthält die erforderlichen Zeilen. Dazu fügen Sie dann noch die Anweisung

```
include public.inc
```

vor die letzte Zeile (»end start«) in »afdload.asm« ein. Speichern Sie die Datei, verlassen Sie den Editor und assemblieren Sie das Programm mit Hilfe der oben empfohlenen Batch-Datei zu einem Com-Programm. In der neu erzeugten Map-Datei steht jetzt das Symbol »Init« unter den als »public« definierten Symbolen. Bei der vom TASM erzeugten Map-Datei sind natürlich auch alle anderen Marken vertreten. Laden Sie nach dem Debugger

```
afdpro afdload.com
```

wieder die Map-Datei:

```
ls afdload.map
```

In der ersten Zeile des Disassembler-Fensters ändert sich sofort die Sprungadresse auf »jmp Init«. Das ist allerdings (beim MASM) bislang das einzige verwendete Label. Wenn Groß- und Kleinschreibung unterschieden werden soll, erreichen Sie das mit der zusätzlichen Anweisung

```
sy /c on
```

Damit beim MASM auch alle anderen Symbole verwendet werden, geben Sie

```
ls afdload.lst
```

ein, wonach der AFD-Pro alle weiteren Symbole aus der Lst-Datei liest. Dieser Schritt ist für TASM-Anwender natürlich nicht mehr erforderlich. Der Sy-Befehl ohne Parameter zeigt alle verwendeten Labels im unteren Bildschirmdrittel (Bild 1). Dieses Fenster läßt sich mit der Taste [Alt-F7] bis zur dritten Zeile des Disassembler-Fensters nach oben vergrößern und mit [Alt-F8] wieder verkleinern. Die Ctrl-Taste in Verbindung mit [F7] bis [F10] verschiebt das Fenster in der Symboltabelle gemäß der angezeigten Richtung in der untersten Zeile des Bildschirms.

Dies ist der angemessene Zeitpunkt, um ein Tastaturmakro festzulegen. Es wäre tatsächlich ziemlich aufwendig, nach jedem Start des Debuggers alle diese Einstellungen vornehmen zu müssen, nur damit die Labels korrekt angezeigt werden. Damit Sie die Definition des Makros dort beginnen, wo Sie das Makro dann auch anwenden, verlassen Sie den AFD-Pro, um ihn per

```
afdpro afdload.com
```

erneut zu starten. Geben Sie nun »xt« ein, um die Makro-Definition zu beginnen (Bild 2). Jede Taste, die Sie von nun an drücken, speichert der AFD-Pro – Sie sollten sich demzufolge bemühen, die folgenden Eingaben korrekt nachzuvollziehen.

Drücken Sie zweimal [F3]. Sie sehen die Dummy-Eingabe »L afdload.com«. Drücken Sie dreimal [Backspace], um die Erweiterung »com« durch »map« zu ersetzen. Mit der

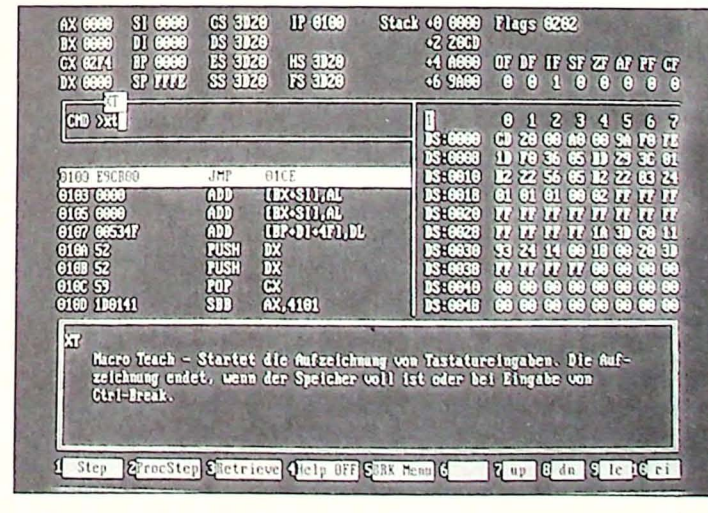


Bild 2. Der AFD-Pro hat eine gelungene Online-Hilfe (unterer Bildschirmteil)

Home-Taste gelangen Sie an den Anfang der Befehlszeile und mit [Cursor-rechts] auf das Leerzeichen neben dem L-Befehl. Die Taste [Ins] schaltet auf den Einfügemodus. Nachdem Sie dem »L« ein »s« angehängt haben, muß die Befehlszeile folgendermaßen aussehen:

```
Ls afdload.map
```

Ist das der Fall, schalten Sie mit [Ins] den Einfügemodus wieder ab und bestätigen die Zeile mit [Enter]. Wie bereits erwähnt, sind bei Verwendung des TASM die folgenden Eingaben bezüglich der Lst-Datei nicht sinnvoll. Drücken Sie [F3], um diesen Befehl zurückzuholen. Modifizieren Sie ihn zu

```
Ls afdload.lst
```

und bestätigen Sie auch diese Zeile mit [Enter].
Wenn Sie die Groß- und Kleinschreibung generell unterschieden wissen wollen, folgt als letztes die Eingabe

```
sy /c on
```

[Ctrl-Break] beschließt die Makrodefinition. Speichern Sie das Makro mit:

```
xw sym.mak
```

Wann immer Sie von nun an die Symbolbearbeitung für ein Programm aktivieren möchten, geben Sie entweder nach »L Dateiname« oder sofort nach dem Aufruf »afdpro Dateiname« (oder später mit dem korrigierten Programm »afdload Dateiname«) das Kommando

```
xx sym.mak
```

ein. Selbstverständlich muß sich die Makrodatei dazu im aktuellen Verzeichnis aufhalten, andernfalls haben Sie den Pfad mitanzugeben.
Die Fehlermeldung des Debuggers nach dem Start von »afdload« ist eigentlich nur ein Schönheitsfehler, dessen Beseitigung warten kann. Viel wichtiger ist die Frage, warum die Eingabe eines Nicht-AFD-Pro-Befehls zum Ausstieg führt. Entweder ist der Interrupt 60hex, der in diesem Fall verwendet wird, nicht korrekt gesetzt, oder die Interrupt-Routine selbst ist ab dem Label »Einsprung« fehlerhaft. Das läßt sich sehr leicht herausfinden: Starten Sie das Programm, und vereinbaren Sie gleichzeitig eine Unterbrechung zu dem Zeitpunkt, an dem der Interrupt 60hex gerade eine neue Adresse erhält. Das ist exakt bei Label »ohne60« erreicht, zu dem der Installationsteil verzweigt, wenn »afdload« bereits im Speicher ist und der Interrupt seine Adresse behalten soll. Verwenden Sie den G-Befehl mit den Start- und Abbruchadressen:

```
g Init,ohne60
```

Nach der Break-Meldung müßte der Interrupt 60hex auf das Label »Einsprung« zeigen. Die Interrupt-Adressen sind in ihrer Reihenfolge im Segment 0 durch 4-Byte-Zeiger (Offset und Segment) vertreten. Der Zeiger auf Interrupt 0 steht bei Offset 0, der von Interrupt 1 bei Offset 4 und so weiter. Weisen Sie AFD-Pro an, im Speicherfenster 1 die Werte ab Offset 60hex x 4 des Segments 0 zu zeigen:

```
m1 0:4x60
```

Tatsächlich errechnet der Debugger das Ergebnis der Multiplikation und stellt im Fenster rechts neben der Kommandozeile die Byte-Inhalte ab HS:0180hex dar, wobei das fiktive Hilfssegment HS zuvor den Wert 0 bekommen hat. Verwenden Sie nun den Sy-Befehl ohne Parameter, und blättern Sie mit [Ctrl-PgUp] und [Ctrl-PgDn] in der Label-Tabelle, bis Sie »Einsprung« finden. Die ersten 4 Byte aus dem Speicherfenster müssen – byteweise von rechts nach links gelesen – exakt die für »Einsprung« geltende Segment- und Offset-Adresse sein. Das ist der Fall, der Interrupt ist also korrekt verbogen. Nun kann der Fehler nur noch in der Interrupt-Routine selbst liegen. Um diese zu prüfen, schalten Sie das Disassembler-Fenster auf das Label »Einsprung«:

```
d Einsprung
```

Bevor Sie die Interrupt-Routine testen, müssen Sie in den Registern sinnvolle Werte ablegen

Die Interrupt-Routine geht davon aus, in den Registern bestimmte Werte zu erhalten. Bevor Sie beginnen, sie Schritt für Schritt nachzuvollziehen, müssen Sie den betreffenden Registern die erforderlichen Werte zuweisen. Zunächst benötigen Sie Speicher für den simulierten Eingabe-String. Dazu bietet sich der Freiraum an, der für Anfügungen an die Kommandozeile zum Aufruf des Debuggers reserviert ist. Das Label heißt »Platz«. Lassen Sie diesen Freiraum im unteren Speicherfenster ausgeben:

```
m2 Platz
```

Dort bringen Sie einen Befehl unter, den AFD-Pro sicher nicht kennt, zum Beispiel »ee«. Das geschieht mit dem Patch-Befehl

```
p Platz,'ee'
```

Das Register SI muß nun die Offset-Adresse von »Platz« erhalten. Diese Adresse steht nach dem Doppelpunkt in der ersten Zeile des Speicherfensters 2:

```
r SI=314
```

Das Datensegment soll auf die Segmentadresse des Befehls zeigen – dies ist ohnehin der Fall. ES muß auf das Segment der Benutzerroutine zeigen. Tut es das nicht, so muß es mit

```
r ES=CS
```

zugewiesen werden. Von DI erwartet das Programm keine Werte, das Register kann unberücksichtigt bleiben. Nur CX muß noch die Anzahl der Zeichen des Befehls bekommen:

```
r CX=2
```

Beginnen Sie nun, den Programmablauf mit [F1] nachzuvollziehen. Der Vergleich in Offset 189hex setzt das Carry-Flag nicht, weil in diesem Fall CX nicht größer ist als AX (vier Zeichen). Der Branch-Befehl in 18Bhex verzweigt also zur Sprungmarke »Fehler«. Dort bekommt DX die Offset-Adresse des Fehlertextes, und die Bearbeitung wird bei Label »Ende« fortgesetzt. SI wird wieder vom Stapel gezogen, damit AFD-Pro weiß, daß der Fehler sich bereits an der ersten Stelle des Befehls bemerkbar gemacht hat.
Aber halt: Auf dem Stapel ist das SI-Register nicht vermerkt. Es wurde zu Beginn der Routine vergessen, dieses Register auf den Stapel zu schieben. Bei einer solchen Verschiebung geht natürlich die ebenfalls auf dem Stapel abgelegte Rücksprungadresse verloren, und der Rechner findet sich erfahrungsgemäß im Daten-Nirwana wieder. Verlassen Sie also den Debugger, und fügen Sie als Zeile 28 in »afdload.asm« den Befehl

```
push SI
```

ein, speichern und assemblieren Sie das Programm bis zur Com-Datei. Rufen Sie wiederum

```
afdload
```

auf. Die »häßliche« Meldung zu Anfang übergehen Sie mit [Esc]. Geben Sie nun »ee« ein. Zweifelsohne wurde versucht, eine Fehlermeldung anzubringen. Zu sehen ist allerdings nicht viel. Die ausgegebenen Zeichen können sehr unterschiedlich sein, es hängt davon ab, was an der Position, auf die der Ausgabezeiger momentan hindeutet, auf Ihrem Rechner gespeichert ist. Sind mindestens zwei oder gar drei Zeichen ausgegeben worden, können Sie diese im Speicher Ihres Rechners suchen. Vielleicht erfahren Sie so, was schiefgelaufen ist.

```
s 0:0,'Zeichen ...'
```

Wenn Sie Glück haben, wird der Rechner diese Zeichen bereits im Segment 0 – irgendwo zwischen den Zeigern – unter Umständen sogar mehrmals finden. Andernfalls lagen zwischen den Zeichen unsichtbare Sonderzeichen, und er findet gar nichts. Wenn Sie die Interrupt-Routine nach obigem Muster noch einmal durchgehen, werden Sie bemerken, daß eigentlich alles korrekt abläuft. Es gibt somit nur noch die Möglichkeit, daß die Register nicht mit den

Werten geladen sind, die sie laut Handbuch haben müssen, oder die Routine darf die Registerwerte nicht so übergeben, wie es das Handbuch verlangt. Die erste Vermutung ist leicht zu überprüfen. Ermitteln Sie dazu die Adresse des Interrupt 60hex:

```
m1 0:4*60
```

Wie bereits erwähnt, geben die ersten vier Byte dieser Adresse (0:180hex) von rechts nach links sowohl das Segment als auch den Offset der Einsprungroutine wieder. Lassen Sie sich die Routine im Disassembler-Fenster anzeigen. Die Segmentangabe müssen Sie dem Speicherauszug im Speicherfenster 1 entnehmen, weil sie von der Speicherbelegung Ihres Rechners (residente Programme, Treiber etc.) abhängt.

```
d <Segment>:185
```

Begeben Sie sich mit dem A-Befehl (ohne Parameter) in den Assemblermodus. Geben Sie exakt das kleine Assemblerprogramm aus Listing 5 ein, wobei Sie statt des Platzhalters »Datensegment« den augenblicklichen Wert des Registers DS übergeben – Sie entnehmen ihn der Registertabelle auf dem Bildschirm links oben.

Dieses Segment ist nämlich das erste unbenutzte, das AFD-Pro Ihnen zur Verfügung stellt. Mit der Kombination [Ctrl-Enter] gelangen Sie wieder in die Kommandozeile. Haben Sie vorhin an der Suche nach der unkenntlichen Fehlermeldung teilgenommen, sollten Sie das Speicherfenster 2 wieder zurücksetzen:

```
m2 ds:0
```

Verursachen Sie jetzt einen Fehler durch die Eingabe des nichtexistenten Ee-Befehls. Sofort ändern sich die Werte im Speicherfenster 2. Schon das Wort ab Offset 0 sticht ins Auge. Es ist das abgelegte Register ES. Es sollte zusammen mit DI auf die Benutzerroutine zeigen. Das Wort ab Offset 8 ist der Inhalt des DI-Registers. ES:DI zeigt auf 0:180hex. Dieses Registerpaar ist nicht der Zeiger auf die Benutzerroutine (wie es im Handbuch vermerkt ist), vielmehr zeigt es auf den Zeiger, der auf die Benutzerroutine hinweist. Das bedeutet, daß das unveränderte Segment ES am Ende auf das Null-Segment gezeigt hat und alle Arbeiten damit ohnedies niemals hätten richtig sein können.

Im Assembler-Listing muß ES den Wert des Codesegments bekommen. Dazu bemühen Sie wieder Ihren Editor und fügen folgende Zeilen als Zeile 28 und 29 in »afdload.asm« ein:

```
push CS
pop ES
```

Assemblieren und linken Sie das Programm und rufen Sie es ohne Parameter auf. An die dann erscheinende Fehlermeldung werden Sie sich noch gewöhnen.

Drücken Sie [Esc] und geben Sie den Ee-Befehl ein, der einen Fehler auslösen soll. Doch es geschieht nichts. »afdload« verhält sich genau wie zuvor, als Sie das Interrupt-Programm testweise überschrieben haben. Vielleicht bemerken Sie, daß sich die Segmentadresse, von der an AFD-Pro Ihnen Speicher zugesteht, um wenige Paragraphen (je 16 Byte) nach unten verschoben hat. Es scheint also so, als ob sich das Programm nicht mehr aus dem Speicher bewegt, wenn Sie den Debugger verlassen.

Das Programm bleibt stets resident, Sie müssen jedoch den Rechner vor dem Start booten

Ermitteln Sie die Adresse, auf die der »Int60« zeigt – sie steht bei 0:180hex. Wenn Sie den D-Befehl mit dieser Adresse ausführen, sehen Sie das von Ihnen vorher eingegebene Testprogramm. Sie sollten den Computer booten und »afdload« erneut aufrufen, um zu testen, ob die bisherigen Änderungen am Quellcode erfolgreich waren. Nach dem Aufruf von »afdload« verwenden Sie wieder den Ee-Befehl. Diesmal wird die korrekte Fehlermeldung (»Befehl unbekannt«) ausgegeben. Daß es diejènige Ihrer Interrupt-Routine ist, erkennen Sie daran, daß ihr das Größer-Zeichen nicht voransteht. Auf die Eingabe »sorry« sollte AFD-Pro antworten: »Vergeben und vergessen..«. Auf die Eingaben »arghl« und »hilfe« folgen ebenfalls die vereinbarten Meldungen. Da hier offenbar alles funktioniert, sollten Sie sich auf die Suche nach dem Fehler machen, der dafür verantwortlich ist, daß das Programm stets resident bleibt. Laden Sie es also mit

```
L afdload.com
```

und rufen Sie das Tastaturmakro auf:

```
xx sym.mak
```

Benutzen Sie diesmal [F2], um das Programm schrittweise zu verfolgen. (Mit [F1] würden Sie jeden Interrupt-Aufruf bis in die tiefsten Urwälder des BIOS verfolgen.)

Arbeiten Sie sich bis zu Offset 1F8hex durch. Dort wird das Register AX mit dem Wert der Adresse 64hex im Segment ES geladen. ES hat die Segmentadresse des augenblicklichen Interrupt 60hex von der vorangegangenen MS-DOS-Funktion 35hex erhalten. Hier wird geprüft, ob »afdload« schon da war. Sie können sich den Quelltext mit seinen Bemerkungen ansehen, wenn Sie

```
br afdload.asm
```

eingeben. Mit dem sich öffnenden Fenster können Sie genauso verfahren, wie mit dem der Symboltabelle, allerdings verschwindet es immer, wenn Sie einen Befehl eingeben. Nach der Eingabe des Br-Befehls (ohne Parameter) erscheint es wieder.

Dort sehen Sie in den Zeilen 78 und 79 die entsprechenden Kommandos. Hier ist erkennbar, worauf diese beiden Anweisungen abzielten. Die ersten zwei Byte des Sprungbefehls zur Init-Routine wurden im Segment des Interrupt 60hex mit dem des momentanen Programms verglichen. Sind sie gleich, ist »afdload« bereits im Speicher. Leider wurde hinter der Angabe »100« die hexadezimale Definition vergessen, daher installiert sich das Programm auch, wenn es bereits vorhanden ist.

Trotzdem ist noch nicht klar, warum das Programm in jedem Fall resident bleibt. Hier hilft nur einschlägige Lektüre oder Erfahrung weiter, die über den Funktionsaufruf 4Bhex weiß, daß nach der Rückkehr kein einziges Register mehr ist, wie es war. Deshalb können Sie den Aufruf mit dem AFD-Pro auch nicht verfolgen, denn kehrt er zurück, zeigt das Disassembler-Fenster irgendwo in die Daten-Prärie. Nicht einmal der Stack ist mehr ansprechbar, man müßte vor dem Aufruf die Register SS und SP explizit zwischenspeichern – natürlich nicht auf dem Stack.

Bislang geht das Programm davon aus, daß das Datensegment DS korrekt zurückkommt. Verlassen Sie hier den Debugger, und editieren Sie den Assemblerquelltext. Verbessern Sie die Zeilen 78 und 79, indem Sie der Offset-Angabe »100« die Bezeichnung »H« anfügen. Die Zeilen lauten dann:

```
mov AX,word ptr ES:100H
mov CX,word ptr CS:100H
```

Außerdem müssen Sie nach dem Funktionsaufruf 4Bhex Ihr Datensegment zurückgewinnen. Fügen Sie als Zeilen 148 und 149 folgende Befehle ein:

```
push CS
pop DS
```

Die Fehlerroutine sollte dann natürlich auch mit einem korrekten Datensegment arbeiten. Fügen Sie die letzten beiden Befehle ebenfalls in den Zeilen 170 und 171 ein. Das Label »CallError« sollte auf jeden Fall in Zeile 170 bleiben, verschieben Sie lediglich den nebenstehenden Befehl (»cmp AX,8«) um die entsprechende Zeilenzahl nach unten. Er steht dann genau unter den von Ihnen eingefügten Zeilen. Nach dem Speichern, Assemblieren und Linken (etc.) müssen Sie zunächst booten. Sie haben nämlich einen nicht funktionierenden

»afdload« im Arbeitsspeicher – und damit können Sie die neue Version nicht testen.

Ist der Rechner wieder bereit, rufen Sie »afdload« auf und verlassen den Debugger wieder. Rufen Sie nun »afdpro« auf und zwar mit »/U60« und einem oder mehreren Leerzeichen als Parameter.

```
afdpro /U60 [Leerzeichen]
```

AFD-Pro startet nun mit der gleichen Fehlermeldung, als ob er von »afdload« geladen worden wäre, und zwar nur wegen des oder der überzähligen Leerzeichen. Drücken Sie [Esc] und prüfen Sie, ob der Zeiger auf den Interrupt 60hex (0:180hex) in den »bewohnten« Arbeitsspeicher sieht oder ob er mit dem Standardwert der nachfolgenden Vektoren identisch ist. Im zweiten Fall ist »afdload« wunschgemäß nicht mehr im Speicher. Nehmen Sie hierbei Falscheingaben vor, wird mangels geladener Fehlerroutine durch Interrupt 60hex ein Systemabsturz provoziert.

Verlassen Sie den Debugger und editieren Sie den Assemblerquelltext. Ab Zeile 98 wird davon ausgegangen, daß, wenn die Kommandozeile leer ist, nichts passieren soll. Andernfalls wird die Zeile in die neue Kommandozeile für den Tochterprozeß (für »command.com«) kopiert, die bislang nur aus Leerzeichen besteht. In jedem Fall enthält die Kommandozeile das Enter-Zeichen (ASCII 13). Dieses Enter-Zeichen muß also immer in die neue Kommandozeile kopiert werden. Dazu löschen Sie die Zeilen 98 und 99, sie werden nicht mehr gebraucht, denn auch eine leere Kommandozeile ist jetzt immer ein Zeichen länger als ausgelesen, da das Enter-Zeichen noch hinzukommt. Als Zeile 98 fügen Sie demzufolge ein:

```
inc CL
```

Das nächste Label (»noCmdLine«) ist nun nutzlos. Entfernen Sie es, speichern Sie den Quelltext und assemblieren Sie diesen. Wenn Sie den TASM verwenden, müssen Sie das Label vor dem Assemblieren auch noch aus der Include-Datei (»public.inc« als Listing 6) streichen.

Starten Sie darauf »afdload«, und freuen Sie sich, daß die Fehlermeldung ausbleibt. Testen Sie das Programm durch »quit r« und erneute Starts – »afdload« wird immer nur einmal im Speicher sein. Es gibt Software (zum Beispiel »Mapmem«), die Ihnen eine Übersicht über den Speicherzustand hinsichtlich residenter Programme zeigt. Solche Programme lassen sich verwenden, um zu prüfen, ob »afdload« sich wieder korrekt entfernt. Haben Sie den Debugger einmal resident verlassen, bleibt die Interrupt-Routine immer im Speicher. Fühlen Sie sich hiermit aufgerufen, ein Programm zu verfassen, das diesen Speicher (MS-DOS-Funktion 49hex) und den Interrupt 60hex wieder freigibt.

Am Ende ein Wort zu der Pascal-Version von »afdload«. Naturgemäß verbraucht sie wesentlich mehr Speicher als das Assemblerprogramm. Allerdings ist die Programmierung damit auch ein wenig leichter, vor allem in Hinblick auf spätere (sinnvollere) Erweiterungen.

Die Funktion »SchonDaTest« ist erklärungsbedürftig. Echte Konstanten, die im Programmteil (nicht im Vereinbarungsteil) auftauchen, legt Turbo Pascal ab Offset 0 im ersten Programmsegment ab. In diesem Fall ist das zuerst ein Byte mit dem Wert 7 und dann die Zeichenkette »Kennung«. Die Funktion legt genau an die Segmentadresse, auf die der Interrupt 60hex zeigt, bei Offset 0 einen String (Länge, Inhalt) ab. Ist sein Inhalt »Kennung«, befindet sich »afdload.exe« bereits im Speicher.

(Dieter Bäumler/al)

Info: »Afd-Pro Full-Screen-Debugger«, AdTec GmbH, 7039 Weil im Schönbuch

```
[BZ3]    1: ; Programm: afdload.asm
[S84]    2: ; Autor: Dieter Bäumler
[6Z6]    3: ; (c)1989 Redaktionsbüro Everts & Hagedorn
[UX3]    4: ;          DOS International
[W85]    5: assume CS:CODE, DS:CODE, ES: CODE
[H22]    6: CODE segment
[A61]    7: org 1ØØH
[ØØØ]    8:
[KH2]    9: Start: jmp Init
[ØØØ]   1Ø:
[7H2]   11: Kennung = ØaaaaH
[AU2]   12: Ablage   db 5 dup (?)
[ZG1]   13: Anzahl = 3
[C53]   14: Befehle  db 'SORRY'
[NZ1]   15:          dw Antwort1
[YE1]   16:          db 'ARGHL'
[TZ1]   17:          dw Antwort2
[NE1]   18:          db 'HILFE'
[XZ1]   19:          dw Antwort3
[BK6]   2Ø: Antwort1 db 'Vergeben und vergessen..',Ø
[956]   21: Antwort2 db 'Es wird schon werden..',Ø
[XX5]   22: Antwort3 db 'Hilf',27h,' Dir selbst, '
[RW3]   23:          db 'so hilft Dir Gott ! ',Ø
[N55]   24: FehlMeld db 'Befehl unbekannt',Ø
[ØØØ]   25:
[CN5]   26: Einsprung: sti     ;Sicher ist sicher
[F94]   27:            mov AX,4   ;wenn der Befehl
[MØ4]   28:            cmp AX,CX  ;weniger als 5
[DK4]   29:            jnc Fehler ;Chr's hat, Fehler
[GN4]   3Ø:            mov BX,5 ;Buchstabenzähler
[2D4]   31:            lea DI,Ablage      ;Die ersten
[9L5]   32: gross:     mov AL,[BX+SI-1]   ;5 Chr's
[134]   33:            and AL,not 2ØH     ;werden in
[HW4]   34:            mov ES:[BX+DI-1],AL ;großen
[6H4]   35:            dec BX ;Buchstaben abgelegt
[BL1]   36:            jne gross
[AZ4]   37:            push CS ;DS nach Hause führen
[6T1]   38:            pop DS  ; ...
[7T4]   39:            mov DX,Anzahl ;Anzahl Befehle
[R64]   4Ø:            lea AX,Ablage  ;Konstanten
[EX3]   41:            lea BX,Befehle-7 ;ablegen
[ULØ]   42:            cld
[DØ3]   43: Naechster: add BX,7
[KØ4]   44:            mov DI,BX ;Konstanten an
[T74]   45:            mov SI,AX ;Index-Register
[KH4]   46:            mov CX,5  ;Anzahl Buchstaben
[YM4]   47:            repe cmpsb ;vergleiche Wort
[RX1]   48:            je Gefunden
[874]   49:            dec DX    ;nächster Befehl
[A92]   5Ø:            jne Naechster
```

```
[N66]   51: Fehler:    lea DX,FehlMeld ;keiner mehr ?
[4D1]   52:            jmp Ende
[AA6]   53: Gefunden:  mov DX,ES:[DI] ;Zeiger holen
[8V4]   54: Ende:      pop SI ;auf ersten Char
[514]   55:            mov AL,1 ;Fehler-Kennung
[EMØ]   56:            cli
[DS1]   57:            iret ;raus
[ØØØ]   58:
[7S5]   59: Init:      mov AH,4aH  ;Puffer-Speicher
[LV3]   6Ø:            mov BX,4ØH  ;freimachen
[381]   61:            int 21H
[932]   62:            jnc genugRAM
[VM4]   63: LowOnRAM:  mov AH,9                ;zu
[YP4]   64:            lea DX,SpeicherFehler ;wenig
[A13]   65:            int 21H               ;RAM
[AV3]   66:            jmp NormEnd           ;Ende
[1C6]   67: genugRAM:  mov AX,35ØØH ;Zeiger auf IntØ
[YZ2]   68:            int 21H      ;ablegen
[TE4]   69:            mov word ptr [OldIntØ],bx
[4H1]   7Ø:            mov ax,es
[1U4]   71:            mov word ptr [OldIntØ+2],ax
[BT4]   72:            mov AX,356ØH ;Zeiger auf Int6Ø
[V73]   73:            int 21H      ;erfragen
[4R3]   74:            mov AX,word ptr ES:1ØØ
[4R3]   75:            mov CX,word ptr CS:1ØØ
[UF1]   76:            cmp AX,CX
[PW1]   77:            jne leg6Øab
[ST4]   78:            mov word ptr Check, Kennung
[HF4]   79:            jmp ohne6Ø ;afdload schon da
[WZ5]   8Ø: leg6Øab:   mov word ptr [OldInt6Ø],BX
[H43]   81:            mov AX,ES  ;ablegen
[MT4]   82:            mov word ptr [OldInt6Ø+2],AX
[4N4]   83:            mov AX,256ØH ;Zeiger auf neue
[MB4]   84:            lea DX,Einsprung ;Routine
[HA3]   85:            int 21H          ;setzen
[WØ5]   86: ohne6Ø:    mov AH,62H ;PSP-Segment
[V83]   87:            int 21H    ;ermitteln
[S94]   88:            mov DS,BX  ;Kommando-Zeile
[ZU3]   89:            push CS    ;dort heraus
[HN4]   9Ø:            pop ES     ;für den Tochter-
[7Y4]   91:            xor CH,CH  ;prozeß kopieren
[2N1]   92:            mov SI,81H
[M42]   93:            mov CL,[SI-1]
[R81]   94:            cmp CL,Ø
[ØØ2]   95:            je noCmdLine
[LZ1]   96:            lea DI,Platz
[ULØ]   97:            cld
[XS1]   98:            repe movsb
[5X6]   99: noCmdLine: push DS ;Environment-Segment
[F14]  1ØØ:            pop ES  ;ermitteln und im
```

```
[LB4]  1Ø1:              push CS ;neuen PSP für den
[TP4]  1Ø2:              pop DS  ;Tochter-Prozeß
[D63]  1Ø3:              mov SI,2cH ;ablegen
[UH2]  1Ø4:              mov AX,ES:[SI]
[3H1]  1Ø5:              mov ES,AX
[5A4]  1Ø6:              mov word ptr [EnvSeg],AX
[2N4]  1Ø7:              lea AX,Command ;Neue Adresse
[675]  1Ø8:              mov word ptr [ComLine],AX ;der
[6U4]  1Ø9:              mov AX,CS ;Kommandozeile und
[ZT4]  11Ø:              mov word ptr [ComLine+2],AX
[7S4]  111:              mov word ptr [FCB1+2],AX ;die
[P14]  112:              mov word ptr [FCB2+2],AX
[E54]  113:              mov word ptr [FCB1],5cH
[L54]  114:              mov word ptr [FCB2],6cH
[WF4]  115:              mov DI,Ø ;Adressen der alten
[SG4]  116:              mov AX,Ø ;FCB's im neuen PSP
[6S2]  117:              mov BX,Ø ;ablegen
[D76]  118: Schleife:    cmp ES:[DI],AX ;Keine ENV's ?
[3N3]  119:              je Neu ;Ersatz-COMSPEC
[Ø25]  12Ø:              mov CX,8 ;Prüfe ENV-Variable,
[HW3]  121:              lea SI, ComSpec ;ob sie
[ULØ]  122:              cld
[A24]  123:              repe cmpsb ;COMSPEC ist.
[BS4]  124:              je Zeige ;setze Aufruf da hin.
[Y36]  125: suchØ:       cmp ES:[DI],AL ;suche das Ende
[ØS3]  126:              je hierØ       ;dieser ENV-
[ZA3]  127:              inc DI         ;Variable.
[BG1]  128:              jne suchØ
[VW5]  129: hierØ:       cmp ES:[DI],AX ;Keine ENV's ?
[VU3]  13Ø:              je Neu ;Ersatz-COMSPEC.
[BØ1]  131:              inc DI
[W74]  132:              jne Schleife ;nächste ENV
[AZ5]  133: Neu:         lea DX,Ersatz ;setze Aufruf-
[1X4]  134:              jmp Rufe ;Zeiger auf Ersatz.
[Z96]  135: Zeige:       push ES ;setze Aufruf-Zeiger
[WD3]  136:              pop DS  ;auf COMSPEC.
[NG1]  137:              mov DX,DI
[6K2]  138: Rufe:        push CS
[LX3]  139:              pop ES  ;ES:BX zeigt auf
[UØ4]  14Ø:              lea BX,EnvSeg ;neuen PSP
[LP3]  141:              mov AX,4bØØH  ;Load&Run
[381]  142:              int 21H
[GE4]  143:              jc CallError ;Aufruffehler
[BN4]  144:              mov AX,Kennung ;war afdload
[SR4]  145:              cmp Check,Kennung ;schon da ?
[S52]  146:              je NormEnd ;ja
[X64]  147:              mov AX,35ØØH ;ist IntØ noch
[L93]  148:              int 21H      ;der alte ?
[4H1]  149:              mov AX,ES
[SU4]  15Ø:              cmp AX,[word ptr OldIntØ+2]
[743]  151:              jne Resident ;Nein
[MD4]  152:              cmp BX,[word ptr OldIntØ]
[W33]  153:              jne Resident ;dito
[LG3]  154: Alter6Øer:   mov AX,256ØH
[ER4]  155:              mov DX,word ptr [OldInt6Ø]
[TV4]  156:              mov BX,word ptr [OldInt6Ø+2]
[ØS4]  157:              mov DS,BX ;alten Int6Ø wieder-
[PC3]  158:              int 21H   ;herstellen
[AV3]  159: NormEnd:     mov AH,4cH ;und
[CC2]  16Ø:              int 21H    ;raus
[486]  161: Resident:    mov DX, offset Init ;AFD ist
[NE4]  162:              int 27H ;resident, wir auch.
[ØØØ]  163:
[G23]  164: CallError:   cmp AX,8
[RR1]  165:              jne bleibt
[E72]  166:              jmp LowOnRAM
[LS2]  167: bleibt:      mov AH,9
[EC3]  168:              lea DX,AufrufFehler
[8C1]  169:              push CS
[E31]  17Ø:              pop DS
[381]  171:              int 21H
[562]  172:              jmp Alter6Øer
[ØØØ]  173:
[LT1]  174: Check     dw Ø
[DØ2]  175: OldIntØ   dd ?
[N12]  176: OldInt6Ø  dd ?
[5K3]  177: ComSpec   db 'COMSPEC='
[1Ø5]  178: Command   db 7FH,' /c afdpro /U6Ø'
[9H3]  179: Platz     db 7ØH dup (2ØH)
[3C4]  18Ø: Ersatz    db '\COMMAND.COM',Ø
[412]  181: EnvSeg    dw ?
[X22]  182: ComLine   dd ?
[8M1]  183: FCB1      dd ?
[9M1]  184: FCB2      dd ?
[UR3]  185: SpeicherFehler db 7,ØdH
[BC3]  186:                db 'Arbeitsspeicher'
[5Y2]  187:                db ' reicht nicht '
[VF2]  188:                db 'aus !',ØdH,24H
[UH3]  189: AufrufFehler   db 7,ØdH
[2W3]  19Ø:                db 'Umgebungs-Variable '
[ØB3]  191:                db ØafH,'COMSPEC',ØaeH
[FM3]  192:                db ' nicht korrekt !',24H
[TF1]  193: CODE ends
[4K1]  194: end Start
```

Listing 1. Das (fehlerhafte) Programm »afdload.asm« lädt den AFD-Pro, nachdem es den Zeiger 60hex auf eine Befehlserweiterung verbogen hat

```
[H95]    1: E9 CE ØØ ØØ ØØ ØØ ØØ ØØ 53 4F 52 52 59 1D Ø1 41
[4F5]    2: 52 47 48 4C 36 Ø1 48 49 4C 46 45 4D Ø1 56 65 72
[DF5]    3: 67 65 62 65 6E 2Ø 75 6E 64 2Ø 76 65 72 67 65 73
[ND5]    4: 73 65 6E 2E 2E ØØ 45 73 2Ø 77 69 72 64 2Ø 73 63
[HH5]    5: 68 6F 6E 2Ø 77 65 72 64 65 6E 2E 2E ØØ 48 69 6C
[UB5]    6: 66 27 2Ø 44 69 72 2Ø 73 65 6C 62 73 74 2C 2Ø 73
[7F5]    7: 6F 2Ø 68 69 6C 66 74 2Ø 44 69 72 2Ø 47 6F 74 74
[ZC5]    8: 2Ø 21 2Ø ØØ 42 65 66 65 68 6C 2Ø 75 6E 62 65 6B
[ZC5]    9: 61 6E 6E 74 ØØ FB ØE Ø7 56 B8 Ø4 ØØ 3B C1 73 32
[RS5]   1Ø: BB Ø5 ØØ 8D 3E Ø3 Ø1 8A 4Ø FF 24 DF 26 88 41 FF
[NH5]   11: 4B 75 F4 ØE 1F BA Ø3 ØØ 8D Ø6 Ø3 Ø1 8D 1E Ø1 Ø1
[DR5]   12: FC 83 C3 Ø7 8B FB 8B FØ B9 Ø5 ØØ F3 A6 74 ØA 4A
[AL5]   13: 75 EF 8D 16 74 Ø1 EB Ø4 9Ø 26 8B 15 5E BØ Ø1 FA
[AP5]   14: CF B4 4A BB 4Ø ØØ CD 21 73 ØB B4 Ø9 8D 16 A3 Ø3
[6P5]   15: CD 21 E9 F2 ØØ B8 ØØ 35 CD 21 89 1E F8 Ø2 8C CØ
[FE5]   16: A3 FA Ø2 B8 6Ø 35 CD 21 26 A1 ØØ Ø1 2E 8B ØE ØØ
[5P5]   17: Ø1 3B C1 75 Ø9 C7 Ø6 F6 Ø2 AA AA EB 13 9Ø 89 1E
[AL5]   18: FC Ø2 8C CØ A3 FE Ø2 B8 6Ø 25 8D 16 85 Ø1 CD 21
[5R5]   19: B4 62 CD 21 8E DB ØE Ø7 32 ED BE 81 ØØ 8A 4C FF
[TS5]   2Ø: FE C1 8D 3E 18 Ø3 FC F3 A4 1E Ø7 ØE 1F BE 2C ØØ
[TG5]   21: 26 8B Ø4 8E CØ A3 95 Ø3 8D Ø6 Ø8 Ø3 A3 97 Ø3 8C
[RG5]   22: C8 A3 99 Ø3 A3 9D Ø3 A3 A1 Ø3 C7 Ø6 9B Ø3 5C ØØ
[AD5]   23: C7 Ø6 9F Ø3 6C ØØ BF ØØ ØØ B8 ØØ ØØ BB ØØ ØØ 26
[BL5]   24: 39 Ø5 74 1C B9 Ø8 ØØ 8D 36 ØØ Ø3 FC F3 A6 74 17
[PF5]   25: 26 38 Ø5 74 Ø3 47 75 F8 26 39 Ø5 74 Ø3 47 75 DF
[VN5]   26: 8D 16 88 Ø3 EB Ø5 9Ø Ø6 1F 8B D7 ØE Ø7 8D 1E 95
[TS5]   27: Ø3 B8 ØØ 4B CD 21 72 38 ØE 1F B8 AA AA 81 3E F6
[7L5]   28: Ø2 AA AA 74 22 B8 ØØ 35 CD 21 8C CØ 3B Ø6 FA Ø2
[YK5]   29: 75 19 3B 1E F8 Ø2 75 13 B8 6Ø 25 8B 16 FC Ø2 8B
[5V5]   3Ø: 1E FE Ø2 8E DB CD 21 B4 4C CD 21 BA D1 Ø1 CD 27
[TM5]   31: ØE 1F 3D Ø8 ØØ 75 Ø3 E9 FØ FE B4 Ø9 8D 16 C9 Ø3
[E95]   32: ØE 1F CD 21 EB D2 ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
[YD5]   33: 43 4F 4D 53 5Ø 45 43 3D 7F 2Ø 2F 63 2Ø 61 66 64
[565]   34: 7Ø 72 6F 2Ø 2F 55 36 3Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø
[G15]   35: 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø
[G15]   36: 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø
[G15]   37: 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø
[G15]   38: 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø
[G15]   39: 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø
[G15]   4Ø: 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø
[2B5]   41: 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 5C 43 4F 4D 4D 41 4E 44
[F55]   42: 2E 43 4F 4D ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
[VA5]   43: ØØ ØØ ØØ Ø7 ØD 41 72 62 65 69 74 73 73 7Ø 65 69
[8D5]   44: 63 68 65 72 2Ø 72 65 69 63 68 74 2Ø 6E 69 63 68
[SC5]   45: 74 2Ø 61 75 73 2Ø 21 ØD 24 Ø7 ØD 55 6D 67 65 62
[AH5]   46: 75 6E 67 73 2D 56 61 72 69 61 62 6C 65 2Ø AF 43
[WK5]   47: 4F 4D 53 5Ø 45 43 AE 2Ø 6E 69 63 68 74 2Ø 6B 6F
[HN2]   48: 72 72 65 6B 74 2Ø 21 24
```

Listing 2. Der Hexdump von »afdload.com« befähigt Sie, auch ohne Assembler oder Compiler die AFD-Pro-Erweiterung zu nutzen

```
[ZP5]    1: { Programm: afdload.pas                  }
[CQ6]    2: { Autor: Dieter Bäumler                  }
[BD6]    3: { (c)1989 Redaktion Everts & Hagedorn }
[EB5]    4: {          DOS International             }
[D32]    5: {$M 16384,Ø,Ø}
[S62]    6: uses Dos,Crt;
[ØØØ]    7:
[6XØ]    8: const
[NK5]    9: ComLine: ComStr = '/c afdpro /U6Ø';
[GA5]   1Ø: path:    PathStr = '\COMMAND.COM';
[H43]   11: Antwort1: string =
[VF4]   12:   'Vergeben und vergessen..';
[K43]   13: Antwort2: string =
[Y74]   14:   'Es wird schon werden..  ';
[L43]   15: Antwort3: string =
[YU3]   16:   'Hilf'+#$27+' Dir selbst,'+
[TG3]   17:   ' so hilft Dir Gott !';
[DØ3]   18: FehlMeld: string =
[XE3]   19:   'Befehl unbekannt'+#Ø;
[7X2]   2Ø: Befehle:  string =
[WH4]   21:   #Ø+#Ø+#Ø+#Ø+#Ø+'SORRY'+#Ø+'ARGHL'+#Ø+
[861]   22:   'HILFE';
[Ø83]   23: EnvFehler: string =
[WU5]   24:   'Umgebungs-Variable '+#$af+'COMSPEC'+
[B35]   25:   'COMSPEC'+#$ae+' nicht korrekt !';
[734]   26: SpeicherFehler: string =
[EØ6]   27:   'Arbeitsspeicher reicht nicht aus !';
[ØØØ]   28:
[STØ]   29: type
[MS2]   3Ø: Adresse = record
[2S1]   31:             Ofs: word;
[RP1]   32:             Seg: word;
[ZNØ]   33:           end;
[Z22]   34: PSP = record
[S94]   35:           junk: array[Ø..43] of byte;
[392]   36:         EnvSeg: word;
[ZNØ]   37:       end;
[D75]   38: EnvType = array[Ø..4Ø95] of char;
[ØØØ]   39:
[RNØ]   4Ø: var
[LU2]   41:       Regs: Registers;
[LE1]   42:   OldInt6Ø,
[WU2]   43:    OldIntØ: Adresse;
[8V1]   44:          i: integer;
[R52]   45:        tmp: string;
[4U2]   46:    SchonDa: boolean;
```

Listing 3. Das Pascal-Programm »afdload.pas« arbeitet annähernd in gleicher Weise wie das Assemblerprogramm

```
[ØØØ]   47:
[Z65]   48: function SchonDaTest: boolean;
[RNØ]   49: var
[7H3]   5Ø:   KennString: string;
[P66]   51:     KennByte: byte absolute KennString;
[793]   52:     Pruefung: ^string;
[K76]   53:     PruefPtr: Adresse absolute Pruefung;
[MUØ]   54: begin
[H22]   55:   KennByte:=7;
[YU3]   56:   KennString:='Kennung';
[5D2]   57:   Regs.AX:=$356Ø;;
[3Ø2]   58:   MsDos(Regs);
[GW3]   59:   PruefPtr.Seg:=Regs.ES;
[7T2]   6Ø:   PruefPtr.Ofs:=Ø;
[6Z4]   61:   if KennString=Pruefung^ then
[823]   62:     SchonDaTest:=true
[554]   63:   else SchonDaTest:=false
[ZNØ]   64: end;
[ØØØ]   65:
[285]   66: {An diese Stelle muß der Block
[WW5]   67:  "getenv.inc" gelesen werden (^K^R),
[ZV5]   68:  wenn mit Turbo 4.Ø gearbeitet wird.}
[ØØØ]   69:
[XP6]   7Ø: procedure Einsprung (Flags,CS,IP,AX,BX,
[ZU2]   71:                      CX,DX,SI,DI,DS,ES,
[6N1]   72:                      BP: word);
[8X1]   73: interrupt;
[RNØ]   74: var
[CB1]   75:   Eingabe,
[SU2]   76:   Ausgabe: string;
[PM1]   77:       t,i: word;
[8S2]   78:    Fehler: boolean;
[Z81]   79:         y: byte;
[MUØ]   8Ø: begin
[TB2]   81:   Fehler:=false;
[EP2]   82:   for i:=1 to CX do
[MUØ]   83:     begin
[DL4]   84:       t:=Mem[DS:SI+i-1] AND NOT $2Ø;
[4Z5]   85:       Mem[Seg(Eingabe):Ofs(Eingabe)+i]:=t
[ZNØ]   86:     end;
[XØ6]   87:   Mem[Seg(Eingabe):Ofs(Eingabe)]:=CX;
[PC5]   88:   case Pos(Eingabe,Befehle) DIV 6 of
[7M3]   89:     1 : Ausgabe:=Antwort1;
[BM3]   9Ø:     2 : Ausgabe:=Antwort2;
[FM3]   91:     3 : Ausgabe:=Antwort3;
[453]   92:     else Fehler:=true;
[ZNØ]   93:   end;
[TV2]   94:   if NOT Fehler then
[MUØ]   95:     begin
[UR1]   96:       Y:=WhereY;
[7Y4]   97:       Writeln(#$d,#$a,#$a,'    ',Ausgabe);
[ØYØ]   98:       AX:=Ø;
[EZ1]   99:       GoToXY(7,y);
[944]  1ØØ:       while NOT KeyPressed do;
[4MØ]  1Ø1:     end
[ZRØ]  1Ø2:   else
[MUØ]  1Ø3:     begin
[2YØ]  1Ø4:       AX:=1;
[F83]  1Ø5:       DX:=Ofs(FehlMeld)+1;
[UF1]  1Ø6:       ES:=DSeg;
[ZNØ]  1Ø7:     end;
[ZNØ]  1Ø8: end;
[ØØØ]  1Ø9:
[MUØ]  11Ø: begin
[HN3]  111:   SchonDa:=SchonDaTest;
[X13]  112:   if NOT SchonDa then
[MUØ]  113:     begin
[AB2]  114:       Regs.AX:=$356Ø;
[3Ø2]  115:       MsDos(Regs);
[YN3]  116:       OldInt6Ø.Seg:=Regs.ES;
[9P3]  117:       OldInt6Ø.Ofs:=Regs.BX;
[YA2]  118:       Regs.AX:=$35ØØ;
[3Ø2]  119:       MsDos(Regs);
[4G3]  12Ø:       OldIntØ.Seg:=Regs.ES;
[8H3]  121:       OldIntØ.Ofs:=Regs.BX;
[6B2]  122:       Regs.AX:=$256Ø;
[K34]  123:       Regs.DS:=Seg(Einsprung);
[A44]  124:       Regs.DX:=Ofs(Einsprung);
[8Y1]  125:       MsDos(Regs)
[ZNØ]  126:     end;
[PZ3]  127:   tmp:=GetEnv('COMSPEC');
[K24]  128:   if tmp<>'' then Path:=tmp;
[P93]  129:   if ParamCount>Ø then
[S54]  13Ø:     for i:=1 to ParamCount do
[FB5]  131:       ComLine:=ComLine+' '+ParamStr(i);
[H63]  132:   Exec(Path,ComLine);
[KS2]  133:   case DosError of
[9EØ]  134:     Ø: ;
[3H4]  135:     8: Writeln(SpeicherFehler);
[X44]  136:     else Writeln(EnvFehler);
[ZNØ]  137:   end;
[X13]  138:   if NOT SchonDa then
[MUØ]  139:     begin
[YA2]  14Ø:       Regs.AX:=$35ØØ;
[3Ø2]  141:       MsDos(Regs);
[U45]  142:       if NOT (Regs.ES=OldIntØ.Seg) or
[L35]  143:          NOT (Regs.BX=OldIntØ.Ofs) then
[841]  144:           Keep(Ø);
[6B2]  145:       Regs.AX:=$256Ø;
[7T3]  146:       Regs.DS:=OldInt6Ø.Seg;
[4U3]  147:       Regs.DX:=OldInt6Ø.Ofs;
[3Ø2]  148:       MsDos(Regs);
[ZNØ]  149:     end;
[KNØ]  15Ø: end.
```

Listing 3. Das Pascal-Programm »afdload.pas« arbeitet annähernd in gleicher Weise wie das Assemblerprogramm

```
[N67]    1: function GetEnv(Variable: string): string;
[RNØ]    2: var
[LB3]    3:   EnvVars: ^EnvType;
[WW5]    4:    EnvPtr: Adresse absolute EnvVars;
[YG2]    5:   Praefix: ^PSP;
[LW5]    6:    PSPptr: Adresse absolute Praefix;
[S82]    7:      ende: boolean;
[2B1]    8:    tmpEnv,
[4T2]    9:    tmpInh: string;
[MUØ]   1Ø: begin
[D12]   11:   ende:=false;
[1Y1]   12:   Regs.AH:=$62;
[3Ø2]   13:   MsDos(Regs);
[9C2]   14:   PSPptr.ofs:=Ø;
[NF3]   15:   PSPptr.seg:=regs.bx;
[KY4]   16:   EnvPtr.seg:=Praefix^.EnvSeg;
[WB2]   17:   EnvPtr.ofs:=Ø;
[UPØ]   18:   i:=Ø;
[AM2]   19:   while not ende do
[MUØ]   2Ø:     begin
[6S3]   21:       tmpEnv:=''; tmpInh:='';
[VN4]   22:       if EnvVars^[i]<>chr(Ø) then
[MUØ]   23:         begin
[T44]   24:           while EnvVars^[i]<>'=' do
[MUØ]   25:             begin
[P55]   26:               tmpEnv:=tmpEnv+EnvVars^[i];
[6VØ]   27:               i:=i+1
[ZNØ]   28:             end;
[K21]   29:           i:=i+1;
[BP4]   3Ø:           while EnvVars^[i]<>chr(Ø) do
[MUØ]   31:             begin
[A35]   32:               tmpInh:=tmpInh+EnvVars^[i];
[6VØ]   33:               i:=i+1
[ZNØ]   34:             end;
[RX3]   35:           if tmpEnv=Variable then
[MUØ]   36:             begin
[DT2]   37:               GetEnv:=tmpInh;
[FS1]   38:               ende:=true
[4MØ]   39:             end
[HU1]   4Ø:           else i:=i+1;
[4MØ]   41:         end
[ZRØ]   42:       else
[MUØ]   43:         begin
[VZ1]   44:           ende:=true;
[SL1]   45:           GetEnv:=''
[4MØ]   46:         end
[4MØ]   47:     end
[ZNØ]   48: end;
```

Listing 4. Wenn Sie mit Turbo Pascal 4.0 arbeiten, müssen Sie »getenv.inc« in »afdload.pas« einfügen

```
[BC1]    1: push DS
[E34]    2: lea AX,[<Datensegment>]
[1H1]    3: mov DS,AX
[4H1]    4: mov AX,ES
[FV1]    5: mov [Ø],AX
[CV1]    6: mov [2],SI
[R31]    7: pop AX
[VV1]    8: mov [4],AX
[5W1]    9: mov [6],CX
[ØV1]   1Ø: mov [8];DI
[9B1]   11: mov AX,Ø
[GRØ]   12: iret
```

Listing 5. Diese Routine (direkt in den AFD-Pro einzugeben) ersetzt für kurze Zeit den Interrupt-Einsprung – sie legt die Register im Speicher ab

```
[LDC]    1: public Ablage, Alter6Øer, Antwort1, Antwort
            2, Antwort3, Anzahl, AufrufFehler
[TSA]    2: public Befehle, bleibt, CallError, Check, C
            omLine, ComSpec, Command
[1HC]    3: public Einsprung, Ende, EnvSeg, Ersatz, FCB
            1, FCB2, FehlMeld, Fehler, Gefunden
[VMB]    4: public genugRAM, gross, hierØ, Kennung, Low
            OnRAM, leg6Øab, Naechster, Neu
[72C]    5: public NormEnd, noCmdLine, OldIntØ, OldInt6
            Ø, ohne6Ø, Platz, Resident, Rufe
[UL8]    6: public Schleife, SpeicherFehler, Start, suc
            hØ, Zeige
```

Listing 6. »public.inc« lesen Sie mit der Include-Anweisung in »afdload.asm«, sofern Sie mit dem TASM arbeiten

Zum Thema Listing

Ohne Spielregeln geht es nicht

Wenn Sie mit dem Gedanken spielen, uns ein Listing anzubieten, oder wenn Sie zum Kreis derer gehören, deren Angebot bereits einmal dankend abgelehnt wurde, dann sollten Sie jetzt weiterlesen. Sie erfahren hier nämlich, welche Dinge zu beachten sind, um die Chance für einen Abdruck Ihres Listings entscheidend zu erhöhen.

Listings in den Rubriken »Anwendung«, »Utility« und »Tips & Tricks« sind dazu gedacht, Ihnen Anregungen zu geben, wie Sie alltägliche oder auch spezielle Aufgabenstellungen und Probleme programmtechnisch in den Griff bekommen, welche Verfahren anzuwenden sind, und wie Sie dabei am besten vorgehen.

Gerade in den beiden erstgenannten Rubriken haben die Listings eine doppelte Funktion: Zum einen stellen sie eine Variante zur Lösung vorgegebener Aufgabenstellungen dar. Andererseits nutzen viele Leser die Programme nicht in ihrer Gesamtheit, sondern greifen einzelne Programmteile, Funktionen und Prozeduren auf, um sie – in leicht abgewandelter Form – in eigenen Applikationen anzuwenden. Dies gelingt um so leichter, je mehr Informationen und Hinweise Sie zu den einzelnen Programmteilen erhalten.

Leider erhalten wir viel zu oft Einsendungen in der »Vogel-friß-oder-stirb-Form«: »Beiliegend erhalten Sie mein Programm »XYirgendwie«. Viel Spaß!« Zu diesem informativen Schreiben (nicht selten ohne Absender) ereilt uns der Spaß in Form eines wandfüllenden Listingausdrucks oder einer Diskette, die nichts weiter enthält als den Quellcode eines Programms in einer beliebigen Programmiersprache. Für welche Version und welchen Compiler und vor allem zu welchem Zweck die Software geschrieben wurde, bleibt für immer das Geheimnis des Autors.

Liebe Programmautoren, Redakteure sind auch nur Menschen und keine wundersamen Wesen, die in Sekundenschnelle ergründen, was Ihre Gehirne in nächtelanger Arbeit ausgebrütet haben. Geben Sie uns daher eine Chance, das Resultat Ihrer geistigen Anstrengungen richtig zu beurteilen, indem Sie die folgenden Regeln beherzigen.

Ihre Einsendung sollte sich aus drei Teilen zusammensetzen:

- dem Anschreiben,
- Ihrem Programm auf einer mit 360 KByte formatierten Diskette und
- einer Programmbeschreibung (als reine ASCII-Datei und ebenfalls auf der Diskette).

Auf dem Anschreiben und der Diskette geben Sie bitte Ihre vollständige Adresse und Ihre Telefonnummer an; Rückfragen der Redaktion lassen sich so am schnellsten abklären. Führen Sie des weiteren alle auf der Diskette befindlichen Dateien namentlich auf und geben Sie stichpunktartig an, wozu die jeweilige Datei gedacht ist beziehungsweise welchen Inhalt sie hat.

Schließlich benötigen wir noch Angaben über den Sinn und Zweck Ihres Programms, wie man es startet (eventuell zu übergebende Parameter, Hardware-Voraussetzungen und so weiter) und für welche Compiler-Version es geschrieben wurde.

Ihr Programm sollten Sie uns in zweifacher Ausführung auf einer mit 360 KByte formatierten Diskette zukommen lassen: den Quellcode als reine ASCII-Datei und daneben das ausführbare Programm in Form einer Exe- oder Com-Datei. Bei Basic-Listings, die für einen Basic-Interpreter (beispielsweise GW-Basic) gedacht sind, ist die zweite Form natürlich hinfällig. Denken Sie in diesem Zusammenhang bitte daran, beim Speichern der Basic-Programme den Parameter »a« anzuhängen, um den Quellcode als ASCII-Datei zu sichern. Zum Beispiel:

```
save "prog",a
```

Ein sehr wichtiger Punkt ist die maximale Spaltenzahl (Anzahl der Zeichen pro Zeile) Ihres Listings. Der Abdruck eines Listings in der DOS International sieht folgendermaßen aus: Links steht, von eckigen Klammern eingeschlossen, ein dreistelliger Prüfsummencode, es schließt sich die Zeilennummer an, gefolgt von einem Doppelpunkt, und erst dann kommt die Programmzeile.

Für eine solche Programmzeile stehen daher nur noch 43 Spalten zur Verfügung. Alle darüber hinausreichenden Zeichen werden automatisch auf die nächste Zeile umgebrochen. Dadurch geht die Struktur von Programmen verloren (ganz gleich, ob es sich dabei um C-, Pascal-, Basic- oder Assembler-Listings handelt), sie werden unübersichtlich und schwerer lesbar.

Deshalb unsere dringende Bitte: Strukturieren Sie Ihre Programme unter Berücksichtigung einer maximalen Zeilenlänge von 43 Zeichen. Bei Beachtung einiger weniger Punkte wird die Strukturierung Ihres Listings durch diese Vorgabe nicht beeinträchtigt.

- Rücken Sie den Quellcode pro Strukturblock (keine willkürlichen Einrückungen) um maximal zwei Zeichen ein.
- Plazieren Sie Kommentare in eigenen Zeilen, anstatt sie an eine Programmzeile anzuhängen.
- Brechen Sie lange Programmzeilen logisch um, das heißt so, daß die Struktur Ihres Programms sichtbar erhalten bleibt.

Eines ist dabei selbstverständlich: Trotz aller Manipulationen, die Sie an Ihrem Quelltext vornehmen, um der Vorgabe von 43 Zeichen pro Programmzeile zu entsprechen, muß dieses Listing immer noch der Syntax des verwendeten Compilers entsprechen. Mit anderen Worten: Es muß sich ohne (Syntax-) Fehlermeldungen compilieren lassen.

Klärende Worte erleichtern die Umformung von Programmteilen für die eigenen Bedürfnisse

Ein undokumentiertes Programm ist nur eine halbe Sache. Der Umstand, daß ein Programm dem interessierten Leser eine Fülle an Prozeduren und Funktionen für eigene Anwendungen beschert, macht eine ausführliche Dokumentation um so dringlicher. Was helfen die schönsten Programmauszüge, wenn man sie – mangels Information – nicht für die eigenen Zwecke anpassen und umformen kann.

Sie, als der Programmautor, wissen am besten über die Vorzüge, Besonderheiten, Feinheiten und eventuell auch Unzulänglichkeiten Ihres Programms Bescheid. Fassen Sie diese Informationen doch einfach in Form eines Artikels zusammen. Sie erleichtern uns die Arbeit, wenn Sie den Text nicht nur in gedruckter Form, sondern auch als reine ASCII-Datei (ohne irgendwelche Steuerzeichen) auf Diskette Ihrer Sendung beilegen. Vermeiden Sie dabei bitte in Ihrem Text durchgängige Großschreibung, zum Beispiel bei Produktnamen oder Programmbefehlen.

Anregungen bezüglich Art und Stil solcher Artikel können Sie jeder DOS-Ausgabe entnehmen. Auch unsere Redakteure stehen Ihnen für Informationen in dieser Richtung gerne zur Verfügung.

In diesem Sinne: Wir freuen uns auf Ihr Listing!

(zi)

WERK STATT

Die Turbo-Assembler-Werkstatt

Borlands Turbo Assembler TASM bietet dem Programmierer einen Komfort, der bereits an die höheren Programmiersprachen heranreicht. Hinzu kommt, daß sich nur in Assembler die Fähigkeiten eines Computers vollständig ausschöpfen lassen. Nachstehend finden Sie einige Anregungen für die Arbeit mit dem TASM und die Nutzung seiner Fähigkeiten.

Nach wie vor schöpfen nur in Assembler geschriebene Programme die Fähigkeiten des Prozessors vollständig aus, weil hier der gesamte Befehlsvorrat des Prozessors zur Verfügung steht. Somit sind beispielsweise zeitkritische Programme einfach zu verwirklichen.
Zudem ist die Größe des erzeugten Codes von wesentlicher Bedeutung, da die 640-KByte-Grenze unter MS-DOS immer noch vorhanden ist, und Hochsprachen benötigen wesentlich mehr Speicherplatz als in Assembler geschriebene Programme. Hinzu kommt, daß Assemblerprogramme ohne Umwege die einzelnen Systemkomponenten des PC ansprechen.
Oben genannte Gründe führen sehr häufig zu einem Kompromiß: Zeitkritische und systemnahe Prozeduren werden in Assembler programmiert und anschließend in Programme eingebunden, die in Hochsprache formuliert sind.
Möchten Sie in Assembler programmieren, so müssen Sie sich zunächst einmal darüber klar werden, welches Speichermodell Sie verwenden wollen oder sogar müssen. In den Speichermodellen steckt die Aufteilung des zur Verfügung stehenden Speichers und die Größe des Programms.
Die Speichermodelle des TASM sind in Bild 1 zusammengefaßt.
Der Turbo Assembler bietet gegenüber dem MASM hier noch ein weiteres Modell: das Modell »tpascal«. Es ist speziell auf die Belange von Turbo Pascal zugeschnitten, inklusive Parameterübergabe (sie werden in der gleichen Reihenfolge wie bei Turbo Pascal angegeben) und Stack-Bedienung (Setzen des Basiszeigers BP, Rücksetzen des Basiszeigers und Beseitigung lokaler Variablen und Parameter vom Stapelspeicher). Als Beispiel hierzu ein Programm, das zwei Zahlen vertauscht, die als Variablen übergeben werden. Das Pascal-Programm finden Sie in Listing 1, die Assemblerprogramme (einmal »normal« und einmal mit »tpascal«) in Listing 2a und 2b.
Zur Einarbeitung zunächst eine Anleitung, wie Sie die Komponenten zu einem lauffähigen Programm zusammensetzen.
Nachdem Sie den Quelltext geschrieben haben, speichern Sie diesen und verlassen den Editor. Anschließend rufen Sie den Assembler auf. Diesen Vorgang wiederholen Sie so lange, bis der Assembler beim Assemblieren des Quelltextes keine Fehlermeldung mehr erzeugt. Bezogen auf das oben genannte Beispiel sieht der Vorgang so aus:

```
turbo list2a.asm
<Text erfassen und Editor verlassen>
tasm list2a
```

Anschließend tippen Sie den Pascal-Code ab und übersetzen (compilieren) diesen. Damit ist das Programm fertig, da während der Übersetzung des Pascal-Programms der Code der Assemblerroutine eingebunden wird.

Speichermodelle des Turbo Assemblers
(für vereinfachte Segmentanweisungen)

TINY	Daten und Code befinden sich in einem Segment für Com-Dateien (max. 64 KByte)
SMALL	Daten max. 64 KByte Code max. 64 KByte (Code und Daten immer vom Typ »near«)
MEDIUM	Daten max. 64 KByte Code > 64 KByte
COMPACT	Daten > 64 KByte (ein Array max. 64 KByte) Code max. 64 KByte
LARGE	Code > 64 KByte Daten > 64 KByte
HUGE	Code > 64 KByte Daten > 64 KByte (ein Array > 64 KByte)
TPASCAL	erzeugt nur Code- und Data-Segment Anpassung an Erfordernisse von Turbo-Pascal (Parameterübergabe, Unterprogrammaufruf)

Bild 1. Die Speichermodelle des Turbo Assemblers

Der Turbo Assembler führt die Segmentdefinitionen selbständig aus

Nachdem Sie ein Speichermodell festgelegt haben, sind damit automatisch die Segmentdefinitionen (Zuweisung der Segmentregister an die einzelnen Programmsegmente) erledigt, die sonst per Assume-Anweisung (zum Beispiel: »assume ds:data«; Zuweisung des DS-Register an das Datensegment) vorgenommen werden. Der Beginn eines Segments wird beispielsweise mit ».code« oder ».data« bezeichnet. Auch hierzu ein einfaches Beispiel (Listing 3). Da es sich dabei um ein reines Assemblerprogramm handelt, müssen Sie nach dem Assemblieren mit dem Turbo Assembler den gewonnenen Objektcode noch in ein lauffähiges Programm verwandeln. Dies geschieht mit dem Linker »Tlink«:

```
tlink list3
```

Der Linker bereitet den Objektcode so auf, daß er durch den MS-DOS-Kommandoprozessor »command.com« in den Speicher geladen und gestartet werden kann. Hierbei unterscheidet man zwischen Com- und Exe-Dateien. Programme mit der Dateierweiterung »com« haben eine maximale Größe von 64 KByte inklusive Daten (Modell »Tiny«), während Exe-Dateien beliebig groß sein dürfen. Wollen Sie Com-Dateien erzeugen, so müssen Sie beim Linken die Option »/t« angeben. Anschließend läßt sich das erzeugte Programm ausführen. Falls es die Größe Ihres Programms erlaubt, sollten Sie auf jeden Fall eine Com-Datei erzeugen. Da erst zur Zeit der Programmausführung feststeht, wo das Programm im Speicher abgelegt wird, hat der Kommandoprozessor bei Exe-Dateien wesentlich mehr zu tun, weil den verschiedenen Seg-

menten nun die Adressen zugeteilt werden müssen (in der Fachsprache: relokatibler Code). Com-Dateien bestehen dagegen nur aus einem Segment, so daß dieser Aufwand hier nicht entsteht; sie können also schneller geladen werden.

Rufen Sie den Assembler einmal ohne zu übersetzendes Programm auf, und er gibt am Bildschirm eine Liste aller vorhandenen Kommandozeilenoptionen aus. Die Option »/l« erzeugt die Listings, die Sie in den hier wiedergegebenen Bildern 2 und 3 sehen. Die einzelnen Spalten zeigen folgende Informationen: Zeilennummer, Offset zu Beginn des aktuellen Segments, erzeugter Code und Quellbefehl. Die anschließenden Listen zeigen alle Makro-Definitionen, eine Segmenttabelle und eine Symboltabelle. Sollten Sie die Symboltabellen nicht benötigen, so schalten Sie die Erzeugung dieser Tabellen durch die Option »/n« einfach aus.

Bei größeren Programmen ist es interessant zu wissen, in welcher Zeile welches Symbol verwendet wird (dies bezeichnet man als Cross-Referenz). Durch Angabe von »/c« werden die Zeilennummern in die obigen Listen aufgenommen. Da der mitgelieferte Debugger sehr komfortabel ist, sollten Sie ihn auch nutzen. Damit der Assembler die nötigen Informationen bereitstellt, müssen Sie die Option »/zi« angeben. Zu beachten ist dann aber, daß Sie beim Linken die Option »/v« angeben, zum Beispiel:

```
tasm list4 /l/zi/n
tlink list4 /v
td list4
```

Damit Sie die gewünschten Optionen nicht immer explizit angeben müssen, sollten Sie diese in der Konfigurationsdatei »tasm.cfg« speichern. Diese Datei wird vom Assembler zuerst ausgewertet. Sollte sich innerhalb dieser Datei eine Option befinden, die Sie gerade nicht benötigen, so geben Sie eine passende, neue Option beim Aufruf des Assemblers an. Diese überschreibt dann die Vorgabe der Konfigurationsdatei. Zusätzlich erweist es sich als sinnvoll, eine Stapeldatei anzulegen, die den Assembler und Linker nacheinander aufruft:

```
copy con asm.bat [Enter]
tasm %1 /l /zi /n [Enter]
tlink %1 /v [Enter]
[F6][Enter]
```

Sie müssen dann beispielsweise nur noch »asm list4« aufrufen – den Rest erledigt der Computer.

Der Turbo Assembler ist so handlich geworden, daß Sie damit umgehen können wie mit einer Hochsprache. Ein Problem bleibt allerdings bestehen. Es ist wie im täglichen Leben:

```
PROZEDUR.ASM

                1 0000                  .model small
                2 0000                  .stack
                3 0000                  .code
                4
                5 0000                  tausch proc
                6 0000  50                      push ax
                7 0001  8B C3                   mov ax,bx
                8 0003  8B D9                   mov bx,cx
                9 0005  58                      pop ax
               10 0006  C3                      ret
               11 0007                  tausch endp
               12
               13 0007  BB 0005         anfang: mov bx,5
               14 000A  B9 000A                 mov cx,10
               15 000D  E8 FFF0                 call tausch
               16 0010  BB 0011                 mov bx,17
               17 0013  B9 0016                 mov cx,22
               18 0016  E8 FFE7                 call tausch
               19 0019  B8 4C00                 mov ax,4c00h
               20 001C  CD 21                   int 21h
               21                               end anfang


MACRO.ASM

                1 0000                  .model small
                2 0000                  .stack
                3 0000                  .code
                4
                5                       tausch macro
                6                               push ax
                7                               mov ax,bx
                8                               mov bx,cx
                9                               pop ax
               10                               endm
               11
               12 0000  BB 0005         anfang: mov bx,5
               13 0003  B9 000A                 mov cx,10
               14                               tausch
1              15 0006  50                      push ax
1              16 0007  8B C3                   mov ax,bx
1              17 0009  8B D9                   mov bx,cx
1              18 000B  58                      pop ax
               19 000C  BB 0011                 mov bx,17
               20 000F  B9 0016                 mov cx,22
               21                               tausch
1              22 0012  50                      push ax
1              23 0013  8B C3                   mov ax,bx
1              24 0015  8B D9                   mov bx,cx
1              25 0017  58                      pop ax
               26 0018  B8 4C00                 mov ax,4c00h
               27 001B  CD 21                   int 21h
               28                               end anfang
```

Bild 2. Der von den Programmen in Listing 4 und 5 erzeugte Code

Mehr Freiheit wird mit mehr Verantwortung erkauft. Die Folgen eines zwar syntaktisch korrekten, aber logisch nicht einwandfreien Programms in Assembler können wesentlich gravierender sein als bei einem Pascal-Programm. Sie müssen also sehr sorgfältig arbeiten und das Programm in Zweifelsfällen erst mit dem Debugger im Einzelschrittverfahren durchlaufen. Der beim Turbo Assembler neue Idealmodus (neu gegenüber dem Makroassembler von Microsoft) hilft Ihnen hierbei durch strengere Typen- und Syntaxprüfungen.

Um Ihnen eine bessere Dokumentationsmöglichkeit und eine bequeme Handhabung zur Verfügung zu stellen, gibt es Prozeduren und Makros. Prozeduren erweitern gewissermaßen den Sprachumfang um neue Befehle, wie bei Hochsprachen auch. Der Code wird nur einmal erzeugt und

```
     TMACRO.ASM

      1 0000                    .model small
      2 0000                    .code
      3                                 org 100h
      4                         tausch macro  reg1,reg2,reg3
      5                                 push reg1
      6                                 mov  reg2,reg3
      7                                 mov  reg3,reg1
      8                                 pop  reg1
      9                                 endm
     10
     11 0100  BB 0005           anfang: mov bx,5
     12 0103  B9 000A                   mov cx,10
     13                                 tausch ax,bx,cx
1    14 0106  50                        push ax
1    15 0107  8B D9                     mov  bx,cx
1    16 0109  8B C8                     mov  cx,ax
1    17 010B  58                        pop  ax
     18 010C  BB 0011                   mov bx,17
     19 010F  B9 0016                   mov cx,22
     20                                 tausch cx,dx,ax
1    21 0112  51                        push cx
1    22 0113  8B D0                     mov  dx,ax
1    23 0115  8B C1                     mov  ax,cx
1    24 0117  59                        pop  cx
     25 0118  B8 4C00                   mov ax,4c00h
     26 011B  CD 21                     int 21h
     27                                 end anfang
```

Bild 3. Das Ersetzen der Parameter bei Makro-Definitionen

```
gotoxy macro Zeile,Spalte
       Fehler macro
              display 'Bei Gotoxy Zeile'
              display 'und Spalte angeben'
              err
              endm

       ifb <zeile>
           Fehler
       else ifb <spalte>
                Fehler
            else mov dh,zeile
                 mov dl,spalte
                 mov ah,2
                 int 10h
            endif
       endif
       endm
```

Bild 4. Ein Makro-Beispiel mit bedingter Assemblierung

von mehreren Stellen aus genutzt. So erhalten Sie Programme, die mit dem angebotenen Speicher sparsam umgehen.
Makros hingegen erleichtern nur dem Programmierer die Arbeit, werden aber im Programm (während des Assemblierens) bei jedem Aufruf vollständig eingesetzt. Diesen Vorgang können Sie sich wie die Funktion »Suchen und Ersetzen« bei einem Textverarbeitungsprogramm vorstellen. Findet der Assembler einen Bezeichner, der zu einer Makro-Definition gehört, so wird der Programmtext innerhalb des Makros an dieser Stelle eingefügt. Der erzeugte Code enthält also entsprechend der Anzahl der Aufrufe Kopien des Makros. Der Code ist dadurch nicht klein, aber schnell, da die Unterprogrammaufrufe entfallen. Durch beide Methoden gewinnen Sie mehr Übersichtlichkeit im Programm. Ein Beispiel soll dies verdeutlichen:

```
Tausch proc
push ax
mov ax,bx
mov bx,cx
mov cx,ax
pop ax
ret

Tausch macro
push ax
mov ax,bx
mov bx,cx
mov cx,ax
pop ax
endm
```

Bis auf ein paar Schlüsselworte sind die Definitionen absolut identisch. Hinzu kommt, daß Sie Unterprogramme und Makros, die Sie immer wieder benutzen, am besten in einer Datei ablegen. Diese Datei binden Sie dann mit Hilfe der Include-Anweisung in das jeweilige Assemblerprogramm ein. Die Programme in Listing 4 und 5 nutzen die obigen Definitionen. Das Ergebnis der Übersetzung (Bild 2) belegt das vorher Gesagte, wenngleich bei diesem Beispiel der Unterschied in der Codegröße noch nicht zum Tragen kommt.
Makros lassen sich auch mit Parametern versehen, womit sie sehr flexibel werden:

```
Tausch macro reg1,reg2,reg3
push reg1
mov reg1,reg2
mov reg2,reg3
mov reg3,reg1
pop reg1
endm
```

Dieses Makro liefert den gleichen Code (Bild 3) wie das vorhergehende, ist aber wesentlich universeller einzusetzen. Es lassen sich so Aufgaben, die sich nur geringfügig voneinander unterscheiden, über die Parameter steuern. Zusätzlich dürfen Sie die Makros auch verschachteln (ein Makro ruft ein zweites Makro auf...). Als Beispiel betrachten Sie sich schon einmal das später noch zu behandelnde Programm in Listing 6.

Makros und bedingte Assemblierung erhöhen den Programmierkomfort

Richtig zur Geltung kommen die Fähigkeiten aber erst durch die bedingte Assemblierung. Sehen Sie sich dazu das Makro in Bild 4 an.
Durch das sinnvolle Ausarbeiten von Makros wird ein hoher Komfort erreicht, wobei die bedingte Assemblierung eine wesentliche Rolle spielt. Bedingte Assemblierung bedeutet, daß der Assembler die Übersetzung von Programmteilen von einer gestellten Bedingung abhängig macht. Ein Beispiel:

```
ifd 80286
pusha
else
push ax
push bx
push cx
...
endif
```

Die Abfrage »ifd 80286« erzeugt abhängig vom Prozessortyp den optimalen Code. Ein weiteres interessantes Beispiel finden Sie im Handbuch zum Turbo Assembler auf Seite 474. Hier wird ein Makro zur Multiplikation eines Registers mit einer Zahl vorgestellt. Ist die Zahl eine Zweierpotenz, so erfolgt die Multiplikation als Linksverschiebung innerhalb des Registers; ist es keine Zweierpotenz, so wird der Multiplikationsbefehl des Prozessors benutzt.
Auch die von den Hochsprachen bekannten Konstanten sollten Sie in Assemblerprogrammen verwenden (hier heißen sie »Equates«). Sie erleichtern das Lesen eines Programms erheblich.
Das speicherresidente Beispielprogramm (TSR-Programm) fängt den Tastatur-Interrupt ab und belegt nach der Aktivierung die Tasten (Listing 6). In diesem Fall werden die Cursor-Tasten mit Symbolen für Rähmchen (einfach oder doppelt) belegt. Die einfachen Rähmchen schalten Sie durch Betätigen der Tasten [Alt-E] ein, die doppelten durch [Alt-D]. Durch [Alt-A] wird das Programm wieder deaktiviert. Es verbleibt aber im Speicher und läßt sich jederzeit wieder aktivieren.
Von Listing 6 umbelegt werden die Tasten [Home], [Cursor aufwärts], [PgUp], [Cursor links], [Cursor

rechts], [End], [Cursor abwärts], [PgDn], [Ins], [+] und [-] des numerischen Ziffernblocks. In den Zeilen 8 bis 10 und 45 bis 50 sind einige Konstanten deklariert. Sie werden zustimmen, daß das Programm durch die Verwendung dieser Konstanten wesentlich lesbarer wird. Entsprechendes gilt auch für die Makros (Zeilen 12 bis 37). Da bei Com-Dateien das Programm immer an der Adresse 100hex beginnen muß, wird die Zeile 39 notwendig.
Durch die Verwendung der Tabellen in den Zeilen 58 bis 66 ist die Anpassung des Programms an Ihre Bedürfnisse recht einfach. So lassen sich anstelle der vorgeschlagenen Tasten andere umbelegen, falls es Probleme mit Anwenderprogrammen gibt.
Auch zusätzliche Zeichen können Sie einbeziehen. Die Größe der Tabelle wird innerhalb des Programms berechnet, so daß außer der Änderung der Tabelle keine weitere Arbeit auf Sie zukommt. Der allseits bekannte Tastaturtreiber »Keybgr« führt eigentlich auch nichts anderes aus als dieses Programm. Damit Sie ein schnelleres Verständnis für die Assemblerprogrammierung und deren Schwierigkeiten bekommen, empfehlen wir Ihnen, Ihre Programme schrittweise mit dem Turbo Debugger zu verfolgen. Sie haben damit sowohl den Quellcode als auch das erzeugte Programm »im Griff« und erkennen sofort die Änderungen der Register beziehungsweise der betreffenden Speicherstellen.

(Helmut Amshove/zi)

```
[GT2]     1: program tausch;
[5H1]     2: uses crt;
[ØØØ]     3:
[W52]     4: var a,b:word;
[ØØØ]     5:
[777]     6: procedure tauschen(var x,y:word);external;
[U53]     7: {$L tausche2.obj }
[ØØØ]     8:
[MUØ]     9: begin
[DB1]    1Ø:  ClrScr;
[AB4]    11:  writeln('Bitte die erste '+
[NZ2]    12:            'Zahl eingeben: ');
[1H1]    13:  readln(a);
[KC4]    14:  writeln('Bitte die zweite '+
[NZ2]    15:            'Zahl eingeben: ');
[2H1]    16:  readln(b);
[U62]    17:  tauschen(a,b);
[GK3]    18:  writeln ('A:',a,' B:',b);
[KNØ]    19: end.
```

Listing 1. Das Programm »tausch.pas« für die Tauschroutinen aus Listing 2a und 2b

```
[H22]     1: code segment
[EA2]     2:       assume cs:code
[ØØØ]     3:
[DS1]     4: ;Parameter
[SK4]     5: var1 equ dword ptr ss:[bp+8]
[9K4]     6: var2 equ dword ptr ss:[bp+4]
[ØØØ]     7:
[9M1]     8: ;Prozedur
[M33]     9: tauschen proc near
[9P2]    1Ø:         public tauschen
[BG5]    11:         mov dx,ds     ; Datensegment retten
[GE5]    12:         push bp       ; Stackpointer setzen
[PG1]    13:         mov bp,sp
[UG5]    14:         lds si,var1   ; Adresse var1 in ds:si
[CE5]    15:         les di,var2   ; Adresse var2 in es:di
[N74]    16:         mov ax,word ptr es:[di]
[AM3]    17:         mov bx,word ptr [si]
[MK2]    18:         mov es:[di],bx
[A52]    19:         mov [si],ax
[4H1]    2Ø:         mov ds,dx
[131]    21:         pop bp
[LUØ]    22:         ret 8
[X72]    23: tauschen endp
[TT1]    24: code     ends
[4MØ]    25:         end
```

Listing 2a. Die Tauschroutine »tausche1.asm«

```
[XA2]     1: .model tpascal
[FTØ]     2: .code
[ØØØ]     3:
[9M1]     4: ;Prozedur
[DS6]     5: tauschen proc near var1:dword,var2:dword
[9P2]     6:         public tauschen
[BG5]     7:         mov dx,ds     ; Datensegment retten
[UG5]     8:         lds si,var1   ; Adresse var1 in ds:si
[CE5]     9:         les di,var2   ; Adresse var2 in es:di
[N74]    1Ø:         mov ax,word ptr es:[di]
[AM3]    11:         mov bx,word ptr [si]
[MK2]    12:         mov es:[di],bx
[A52]    13:         mov [si],ax
[4H1]    14:         mov ds,dx
[3PØ]    15:         ret
[X72]    16: tauschen endp
[4MØ]    17:         end
```

Listing 2b. »tausche2.asm«: eine weitere Formulierung der Tauschroutine

Software-Autoren gesucht

Haben Sie interessante Programme geschrieben?
Zum Beispiel Anwendungen, die zu umfangreich sind, um als Listing abgedruckt zu werden?
Oder Programmiertools, die mit professionellen Produkten konkurrieren können?
Spiele? Grafikprogramme? Datenbanken? Utilities?

Haben Sie Ideen für neue Software?
Vielleicht haben Sie bereits begonnen zu programmieren, aber scheitern an einem bestimmten Problem?
Oder Sie können aus Zeitmangel nicht alle Ideen realisieren?

Haben Sie Erfahrung als Programmierer?
Suchen Sie eine Aufgabe, die fasziniert? Wollen Sie im Team ein Projekt realisieren?

Zögern Sie nicht, uns Ihre Programme und Ideen zuzusenden. Gern werden wir uns gegebenenfalls mit Ihnen um eine Vermarktung der Produkte kümmern. Wenn Sie interessiert sind, schicken Sie uns die aktuelle Version des Programms auf Datenträger und einen Ausdruck der Dokumentation.

Redaktionsbüro Everts & Hagedorn,
Redaktion DOS International, Stichwort »Software-Autor«, Hans-Stießberger-Str. 3, 8013 Haar

```
[XY1]    1: .model small
[ES1]    2: .stack 1ØØh
[2SØ]    3: .data
[DR5]    4: meldung  db 'Dies ist ein Programm'
[TR3]    5:          db ' mit vereinfachten '
[M44]    6:          db 'Segmentanweisungen. '
[KNØ]    7:          db '$'
[ØØØ]    8:
[FTØ]    9: .code
[DX5]   1Ø: start: mov ax,dgroup  ; Datensegment
[SE3]   11:        mov ds,ax      ; setzen
[KM3]   12:        mov dx,offset meldung
[3M1]   13:        mov ah,Ø9h
[381]   14:        int 21h
[BW1]   15:        mov ax,4cØØh
[381]   16:        int 21h
[4MØ]   17:        end
```

Listing 3. »segment.asm« verwendet vereinfachte Segmentdefinitionen

```
[XY1]    1: .model small
[D21]    2: .stack
[FTØ]    3: .code
[ØØØ]    4:
[XY1]    5: tausch proc
[PC1]    6:        push ax
[BH1]    7:        mov ax,bx
[EH1]    8:        mov bx,cx
[R31]    9:        pop ax
[3PØ]   1Ø:        ret
[XY1]   11: tausch endp
[ØØØ]   12:
[EN2]   13: anfang: mov bx,5
[EE1]   14:         mov cx,1Ø
[9Z1]   15:         call tausch
[UE1]   16:         mov bx,17
[LE1]   17:         mov cx,22
[9Z1]   18:         call tausch
[BW1]   19:         mov ax,4cØØh
[381]   2Ø:         int 21h
[UP1]   21:         end anfang
```

Listing 4. Verwendung einer Prozedur: »prozedur.asm«

```
[XY1]    1: .model small
[D21]    2: .stack
[FTØ]    3: .code
[ØØØ]    4:
[ZØ2]    5: tausch macro
[PC1]    6:        push ax
[BH1]    7:        mov ax,bx
[EH1]    8:        mov bx,cx
[R31]    9:        pop ax
[HPØ]   1Ø:        endm
[ØØØ]   11:
[EN2]   12: anfang: mov bx,5
[EE1]   13:         mov cx,1Ø
[G31]   14:         tausch
[UE1]   15:         mov bx,17
[LE1]   16:         mov cx,22
[G31]   17:         tausch
[BW1]   18:         mov ax,4cØØh
[381]   19:         int 21h
[UP1]   2Ø:         end anfang
```

Listing 5. »macro.asm« zeigt ein Assemblermakro

```
[K44]    1: ; Programmname: TAST.ASM
[WY3]    2: ; (C) 1989 Everts & Hagedorn
[8U3]    3: ;          DOS International
[ØØØ]    4:
[UK4]    5: ; Neuer Tastaturinterrupt
[LB6]    6: ; Ersetzt bei Bedarf die Cursor-Tasten
[7L3]    7: ; durch Liniensymbole
[ZK6]    8: ; Einschalten durch Drücken der Taste
[F92]    9: ; [Scrollock]
[ØØØ]   1Ø:
[625]   11: .model tiny      ; Speichermodell
[4C4]   12: .code            ; für Com-Dateien
[ØØØ]   13:
[NG2]   14: DosInt  EQU  21h
[TM2]   15: Get_Vec EQU  35h
[EN2]   16: Set_Vec EQU  25h
[ØØØ]   17:
[7F3]   18: DosCall macro FunkNr
[182]   19:         mov ah,funknr
[EW1]   2Ø:         int DosInt
[HPØ]   21:         endm
[ØØØ]   22:
[S63]   23: GetVector macro nr
[7G1]   24:         mov al,nr
[ZY1]   25:         DosCall 35h
[HPØ]   26:         endm
[ØØØ]   27:
[G73]   28: SetVector macro nr
[7G1]   29:         mov al,nr
[XY1]   3Ø:         DosCall 25h
[HPØ]   31:         endm
[ØØØ]   32:
```

```
[AE3]   33: PushReg macro liste
[5R2]   34:         irp reg,<liste>
[4E1]   35:         push reg
[HPØ]   36:         endm
[HPØ]   37:         endm
[ØØØ]   38:
[683]   39: PopReg  macro liste
[5R2]   4Ø:         irp reg,<liste>
[WB1]   41:         pop reg
[HPØ]   42:         endm
[HPØ]   43:         endm
[CZ5]   44: ;Einsprungpunkt für Com-Dateien
[A61]   45:     org 1ØØh
[ØØØ]   46:
[NT2]   47: ;Programmanfang
[DØ3]   48: start: jmp install
[ØØØ]   49:
[9S4]   5Ø: ;zu verwendender Interrupt
[RA2]   51: IntNr      EQU 16h
[R65]   52: LiesTaste EQU Ø    ; Funktion Nr Ø
[9N5]   53: Sonder     EQU Ø    ; Sondertastenflag
[AL5]   54: TasteE     EQU 12h ; Taste E gedrückt
[9Ø4]   55: TasteD     EQU 2Øh ; Taste D
[5Ø4]   56: TasteA     EQU 1Eh ; Taste A
[ØØØ]   57:
[562]   58: ;Datenbereich
[RV4]   59: OldInt  dd Ø    ; alter Interrupt
[Z66]   6Ø: OldSs   dw Ø    ; Stack des unterbrochenen
[CU3]   61: OldSp   dw Ø    ; Programms
[ZZ5]   62: aktiv   db Ø    ; Programm bereits aktiv
[616]   63: ;Tabelle der zu ersetzenden Tasten
[P14]   64: tabelle1  db  71, 72, 73, 74, 75
[923]   65:           db  82, 77, 78, 79, 8Ø, 81
[BØ4]   66: ;Tabelle einfache Rahmen
[XA4]   67: TabelleE  db 218,194,191,196,195
[F53]   68:           db 197,18Ø,179,192,193,217
[K34]   69: ;Tabelle doppelte Rahmen
[D74]   7Ø: TabelleD  db 2Ø1,2Ø3,187,2Ø5,2Ø4
[843]   71:           db 2Ø6,185,186,2ØØ,2Ø2,188
[ØØØ]   72:
[ZU5]   73: ofs      dw Ø ; Tabellenoffset einfach
[ZP2]   74:               ;              /doppelt
[ØØØ]   75:
[B13]   76: ; Neuer Interrupt
[786]   77: Neuint: cmp ah,LiesTaste ; soll Taste
[GG4]   78:         je taste         ; gelesen werden
[GG5]   79:         jmp  cs:[oldint] ; nein normaler
[322]   8Ø:                          ; Interrupt
[HV5]   81: taste:  pushf            ; Flags auf Stack
[DB5]   82:         call cs:[oldint] ; Alter Int als
[3T2]   83:                          ; Unterprogramm
[YP5]   84:         PushReg <bx,cx,di> ; benötigte Reg
[BE1]   85:                          ; sichern
[N24]   86:         ; Hotkey zum Aktivieren
[SD2]   87:         ; (Sondertaste)
[R52]   88:         cmp al,Sonder
[A65]   89:         jne l1           ; keine Sondertaste
[562]   9Ø:         cmp ah,TasteE
[ØZØ]   91:         jne l2
[6P5]   92:         mov cs:[aktiv],1 ; einfache Rahmen
[ØN4]   93:         mov ofs,TabelleE-Tabelle1-1
[4D1]   94:         jmp ende
[T13]   95: l2:     cmp ah,TasteD
[3ZØ]   96:         jne l3
[726]   97:         mov cs:[aktiv],2 ; doppelter Rahmen
[ZM4]   98:         mov ofs,TabelleD-Tabelle1-1
[4D1]   99:         jmp ende
[P13]  1ØØ: l3:     cmp ah,TasteA
[XYØ]  1Ø1:         jne l1
[F45]  1Ø2:         mov cs:[aktiv],Ø ; ausschalten
[4D1]  1Ø3:         jmp ende
[XY5]  1Ø4: l1:     cmp cs:[aktiv],Ø ; kein Ersatz
[C61]  1Ø5:         je  ende
[593]  1Ø6:         lea di,cs:[tabelle1]
[6X5]  1Ø7: ; Tabelle1 nach Scancode durchsuchen
[PW5]  1Ø8: ; Tabellengröße in cx als Zähler
[U24]  1Ø9:         mov cx,TabelleE-Tabelle1
[Z33]  11Ø:         PushReg <ax,es,cs>
[G31]  111:         pop es
[6F1]  112:         mov al,ah
[ULØ]  113:         cld
[XZ1]  114:         repne scasb
[S82]  115:         PopReg <es,ax>
[VB1]  116:         jne ende
[TF3]  117:  ;Ersatzzeichen holen
[5Ø2]  118:         add di,[ofs]
[5G2]  119:         mov al,cs:[di]
[XTØ]  12Ø: ende:
[3N2]  121:         PopReg <di,bx,cx>
[GRØ]  122:         iret
[9C4]  123: ; ------------------------------------
[364]  124: Install: GetVector IntNr
[525]  125:          mov word ptr cs:[oldint],bx
[B55]  126:          mov word ptr cs:[oldint+2],es
[HH3]  127:          mov dx, offset Neuint
[XU2]  128:          SetVector IntNr
[PD2]  129:          lea dx,install
[N81]  13Ø:          int 27h
[4K1]  131:      end start
```

Listing 6. »tast.asm« definiert die Tastatur um

ABONNEMENT

DIE QUAL DER WAHL

ABONNEMENTPREISE MIT VORTEIL:

DOS International:
Im Inland bzw. West-Berlin
6 Ausg. = 42,–
12 Ausg. = 82,50

Im europäischen Ausland
6 Ausg. = 55,–
12 Ausg. = 110,–

Im außereuropäischen Ausland
6 Ausg. = 75,–
12 Ausg. = 150,–

DOS Extra
Im Inland bzw. West-Berlin
4 Ausgaben 60,– DM

Im europäischen Ausland
4 Ausgaben 70,– DM

Im außereuropäischen Ausland
4 Ausgaben 80,– DM

DOS INTERNATIONAL

oder

DOS EXTRA

oder beide

Widerrufsrecht: Wir garantieren jedem Abonnenten das Recht, seine Bestellung innerhalb einer Woche nach Abschluß schriftlich zu widerrufen. Die rechtzeitige Absendung des Widerrufsschreibens genügt zur Fristwahrung.

Bitte Bestellkarte benutzen

DMV Daten- und Medienverlag

DMV-VERLAG · POSTFACH 250 · 3440 Eschwege

Von »exe« nach »com« – Ersatz für »Exe2bin«

Zumindest jedem Assemblerprogrammierer ist das Konvertierungsprogramm »Exe2bin«, das im Lieferumfang nahezu jeder MS-DOS-Version enthalten ist, bekannt. Obwohl es oft benutzt wird, wissen doch viele nicht um die genauen Hintergründe und die Funktionsweise dieses Programms. Zu einem besseren Verständnis soll das Programm »exe2com«, das dieselbe Funktion übernimmt wie »Exe2bin« und zudem an keine MS-DOS-Version gebunden ist, beitragen.

Im Prinzip läßt sich das Programm »Exe2bin«, das Exe- in Com-Dateien umwandelt, in jede Programmiersprache einsetzen. Vor allem aber bei der Programmierung in Assembler ist ein Com-Programm für viele Probleme effektiver als eine entsprechende Exe-Prozedur. Fast alle in Assembler entwickelten residenten Programme werden nach dem Com-Modell aufgebaut.

Tatsächlich verlangen auch die wenigsten Assemblerprogramme wirklich ein Exe-Format. Ein Programm als Exe-Datei auszurichten ist nur dann sinnvoll, wenn Programm und Daten zusammen die 64-KByte-Grenze überschreiten oder der Stack mehr als 256 Byte belegt. Letztere Begrenzung des Com-Formats läßt sich durch geschickte Programmierung sogar umgehen.

Schon das Betriebssystem CP/M kannte das Prinzip des Code-Programms

Das Prinzip des Com-Programms ist kein für den PC neu entwickeltes Konzept. Schon unter dem 8-Bit-Betriebssystem CP/M waren Programmdateien als Com-Dateien gespeichert. Das wesentlichste Merkmal eines Com-Programms ist die Tatsache, daß Programme auf der Diskette schon so vorliegen, wie sie auch nachher im Speicher stehen. Das Com-Programm im Speicher ist also ein direktes Abbild des Com-Programms auf der Diskette.

Nun verhält es sich aber so, daß Programme unter MS-DOS bei jedem Aufruf an andere Positionen in den Speicher geladen werden. Je nachdem, wieviel Speicherplatz schon belegt ist, kann ein Com-Programm bei einem Aufruf im oberen Drittel des Speichers stehen, bei einem anderen Aufruf aber im unteren Viertel. Um diese unter MS-DOS unverzichtbare Fähigkeit der Verschiebbarkeit zu gewährleisten, darf ein Com-Programm nur aus einem einzigen Segment bestehen. Beim Start eines solchen Programms sucht das Betriebssystem im Speicher einen freien Speicherblock und lädt das Programm an die dazugehörige Segmentadresse. Alle Bezüge innerhalb des Segments werden dann als relativ zu dieser Segment-Startadresse aufgefaßt.

Da ein Com-Programm aus diesem Grund auf ein einziges Segment begrenzt ist, müssen Daten und Code im gleichen Segment untergebracht und das DS- und CS-Register auf das gleiche Segment initialisiert werden. Beim Start eines Com-Programms legt MS-DOS außerdem in den ersten 256 Byte den Programmsegment-Vorspann PSP an. Hier finden sich Informationen über Kommandozeilen-Parameter, Datenpuffer für die Ein-/Ausgabe und weitere Parameterblöcke. Daher muß ein Programm, um Platz für den PSP zu schaffen, mit einer Anweisung »org 100hex« beginnen.

Das Exe-Format ist eine Spezialentwicklung für den Personalcomputer

Wie man sieht, gibt es eine ganze Reihe an Regeln, die ein Entwickler von Maschinenprogrammen im Com-Format berücksichtigen muß. Um es Ihnen ein wenig einfacher zu machen, finden Sie daher in Listing 1 (»commaske.asm«) das Listing eines Programms, das Sie als Gerüst für Ihre Com-Programme verwenden können. An den im Listing gekennzeichneten Stellen können Sie Ihre Daten und Ihren Quelltext unterbringen.

Im Gegensatz zum Com-Programm wurde das Exe-Format speziell für den PC entwickelt. Ein Exe-Programm ist genau wie ein Com-Programm frei im Speicher verschiebbar, muß sich aber dennoch nicht an die für Com-Programme geltenden Begrenzungen halten. Es lassen sich (fast) beliebig viele Segmente definieren, die Startadresse des Programms ist nicht festgelegt etc. Listing 2 (»exemaske.asm«) zeigt das Gerüst für ein Exe-Programm, wie Sie es als Grundlage für eigene Exe-Programme verwenden können. An den gekennzeichneten Stellen können Sie Ihren Quelltext und Ihre Daten in das Listing einfügen.

Wie beschrieben, ist die Verschiebbarkeit von Programmen, die weniger als 64 KByte lang sind, kein Problem. Jede Adreßangabe innerhalb des Programms wird dabei als relativer Offset zur Segment-Startadresse des Programms verstanden. Ganz anders sieht das aber nun bei Programmen aus, die über mehr als ein Segment verfügen. Hier kann es sein, daß Adreßbezüge segmentübergreifend gelten. Dann ist es aber nicht mehr möglich, den Adreßbezug als relativen Offset zur Programmstartadresse zu betrachten, denn es muß explizit im Programmcode angegeben werden, auf welches Segment sich die Adreßangabe bezieht. Das Problem dabei liegt in der Tatsache, daß die Segmentangaben zu jedem Offset im Programmcode selbst gespeichert sind.

Im »Header« stehen die absoluten Segmentadressen des Exe-Programms

Daher besteht die einzige Möglichkeit, ein solches Programm dennoch frei verschiebbar zu halten, darin, nach dem Laden des Programms diese segmentübergreifenden Bezüge auf die tatsächliche Programmstartadresse umzurechnen. Dazu steht einem Exe-Programm eine Tabelle voran, die alle absoluten Segmentadreßangaben in einer Liste zusammenfaßt. Zu jedem Element der Liste wird in diesem Header gespeichert, wo es sich relativ zum Programmbeginn im Programmcode befindet. Wenn das Betriebssystem nun ein solches Programm lädt, werden zunächst alle in der Tabelle angegebenen Adreßangaben auf die tatsächliche Position des Programms im Speicher umgerechnet. Das heißt, es wird zu jeder absoluten Segmentangabe, die sich auf ein Segment innerhalb des Programms bezieht, die Segmentstartadresse addiert und das Ergebnis wieder im Programmcode gespeichert. Diesen Vorgang nennt man

auch »Relozierung«. Dann werden die Register SS, SP, CS und IP mit Werten initialisiert, die ebenfalls in der Tabelle zu finden sind. Danach wird das Programm gestartet. Tabelle 1 stellt die Informationen, die sich im Exe-Header befinden, zusammen.

Jedes Programm, sei es nun eine Com- oder ein Exe-Datei, wird vom Assembler zunächst in das Exe-Format übersetzt. Für Com-Programme muß man dieses Exe-Format nachträglich wieder in das Com-Format umwandeln. Dazu dient das Programm »exe2bin.exe«.

Und genau diese Aufgabe erledigt auch das Programm »exe2com.exe« in Listing 3. Es arbeitet also – obwohl es ein wenig komfortabler und unhabhängig von der Version Ihres Betriebssystems ist – im Prinzip genauso wie das Exe2bin-Original. Beim Original wurde ein Exe-Programm in ein Com-Programm umgewandelt, das dann aber die Namenserweiterung »bin« trug. In der Regel folgte daher einem Aufruf von »exe2bin« ein Ren-Befehl, der die Bin-Datei in eine Com-Datei umbenannte. Bei »exe2com« werden Programme direkt mit der Namenserweiterung »com« gespeichert, um dieses überflüssige Umbenennen zu vermeiden.

Man kann »exe2com« wie ein gewöhnliches Programm durch Eingabe des Programmnamens mit nachfolgendem [Enter] aufrufen. Dann wird zunächst nach dem Namen des Programms gefragt, das in das Com-Format übersetzt werden soll. Hier können Sie einen Dateinamen oder einen ganzen Pfad eingeben, oder es kann auch ein Laufwerksbuchstabe dem Programm vorangehen. Wird keine Namenserweiterung (das ist der Teil des Dateinamens hinter dem Punkt) angegeben, so hängt »exe2com« automatisch ein »exe« an. Es läßt sich natürlich auch mit anderen Namenserweiterungen arbeiten.

Das Ersatzprogramm »exe2com« ist nicht schwierig in der Handhabung

Ruft man das Programm sofort mit einem Dateinamen in der Kommandozeile auf (zum Beispiel mit »exe2com druck«), dann entfällt die Aufforderung zur Eingabe des Dateinamens, und »exe2com« übernimmt den in der Kommandozeile angegebenen Dateinamen.

Als nächstes wird der Name einer Zieldatei abgefragt. Damit ist die Datei gemeint, in der das Com-Programm nach der Konvertierung abgespeichert werden soll. Geben Sie hier einfach [Enter] ein, so wird der Name der Quelldatei mit der Namenserweiterung »com« verwendet. Sie können natürlich auch hier andere Dateinamen und andere Namenserweiterungen wählen. Auch die Eingabe eines Pfades, eines zusätzlichen Verzeichnisnamens oder eines vorangestellten Laufwerksbuchstabens ist erlaubt.

Anstatt den Namen der Zieldatei einzugeben, können Sie diesen Namen auch gleich als zweiten Parameter in der Kommandozeile angeben. So übersetzt beispielsweise der Aufruf

```
exe2com druck drucker
```

das Programm »druck.exe« in das Programm »drucker.com«.

Sind auf diese Weise die Dateinamen ausgewählt, beginnt die Konvertierung, falls nicht vorher ein Fehler auftritt. Mögliche Fehlerquellen sind dabei, daß die Quelldatei nicht existiert, die Diskette voll ist (die Zieldatei kann also nicht angelegt werden) oder das Programm sich nicht konvertieren läßt (ein als Com-Programm angelegtes Listing muß sich – wie oben beschrieben – an gewisse Konventionen halten, damit es konvertiert werden kann).

(Martin Althaus/hi)

Byte	Bedeutung
00hex-01hex	Müssen die beiden Byte 4D und 5Ahex enthalten.
02hex-03hex	Die Anzahl der Bytes in der letzten Seite der Datei, eine Seite besitzt eine Länge von 512 Byte, Reihenfolge des Anzahlwertes: niederwertig/höherwertig.
04hex-05hex	Größe der gesamten Datei (inklusive Header) in Seiten, der Wert ist in der Reihenfolge niederwertig/höherwertig abgelegt.
06hex-07hex	Anzahl der Einträge in der Relozierungstabelle, Reihenfolge des Wertes: niederwertig/höherwertig.
08hex-09hex	Größe des Headers in 16-Byte-Abschnitten, Reihenfolge des Wertes: niederwertig/höherwertig, auch der letzte Abschnitt, der möglicherweise nicht komplett belegt ist, wird mitgezählt. Die Größe des Headers dient zur Bestimmung des Anfangs des eigentlichen Programms.
0Ahex-0Bhex	Minimale Anzahl der 16-Byte-Abschnitte, die oberhalb des Programms noch als freier Platz verbleiben müssen, in der Reihenfolge niederwertig/höherwertig. Der Linker setzt diesen Wert auf 0, damit Exe-Programme – falls der Speicher schon weitgehend belegt ist – den gesamten Speicherplatz bis nach »ganz oben« benutzen können. Ist der Wert, den diese Speicherstellen ergeben, nicht mehr oberhalb des Programms verfügbar, so wird die Fehlermeldung »memory full« veranlaßt.
0Chex-0Dhex	Maximale Zahl der 16-Byte-Abschnitte, die oberhalb des Programms belegt werden sollen, in der Reihenfolge niederwertig/höherwertig. MS-DOS versucht, den mit diesem Wert spezifizierten Speicherplatz zu reservieren. Schlägt dieser Versuch fehl, so wird der maximal verfügbare Platz reserviert. Der Linker setzt diesen Wert auf FFFFhex. Diese Anforderung schlägt immer fehl, aber zurückgeliefert wird der Platz, der maximal verfügbar ist. Damit werden einerseits die Programme so »tief« wie möglich geladen (weil »oben« so viel wie möglich reserviert wird), andererseits wird der gesamte zur Verfügung stehende Speicher einem Programm zugewiesen.
0Ehex-0Fhex	Wert, der in das SS-Register geladen wird. Dieser Wert liegt in der Reihenfolge niederwertig/höherwertig vor und wird von MS-DOS noch reloziert.
10hex-11hex	Wert, der in das SP-Register geladen wird. Dieser Wert liegt in der Reihenfolge niederwertig/höherwertig vor.
12hex-13hex	Negative Summe aller Bytes in der Datei. Die Bedeutung dieses Wertes ist nicht genau bekannt. Eine Veränderung des Wertes hat keine Auswirkung auf das korrekte Arbeiten des Programms.
14hex-15hex	Wert, der in das IP-Register geladen wird. An dieser Stelle innerhalb des Codesegments beginnt die Programmausführung.
16hex-17hex	Wert, der in das CS-Register geladen wird. Dieser Wert liegt in der Reihenfolge niederwertig/höherwertig vor und wird von DOS vor Programmstart reloziert.
18hex-19hex	Relativer Offset vom Beginn der Relozierungstabelle zum Beginn des Programms.
1Ahex-1Bhex	Die Overlay-Nummer, wie sie vom Linker generiert wird.

Die Tabelle zeigt die Struktur eines Exe-Headers

```
[P3Ø]     1: ;
[7V4]     2: ; Programmname: COMMASKE.ASM
[P3Ø]     3: ;
[K67]     4: ; Gerüst für Assembler-Programme, die als
[D27]     5: ; COM-Programme assembliert werden sollen
[P3Ø]     6: ;
[WY3]     7: ; (C) 1989 Everts & Hagedorn
[8U3]     8: ;          DOS International
[CZ3]     9: ; Autor:  Martin Althaus
[K84]    1Ø: ; letzte Änderung: 12.4.89
[P3Ø]    11: ;
[5U4]    12: ;----------------------------------------
[P3Ø]    13: ;
[SY6]    14: ; In diesem Segment wird der Programmcode
[5K7]    15: ; untergebracht. Alle Programme, Prozeduren
[N87]    16: ; und Unterprogramme können Sie in dieses
[BU2]    17: ; Segment setzen.
[P3Ø]    18: ;
[877]    19: ;Segment soll an einem Paragraphen beginnen
[PN4]    2Ø: programm  segment para 'code'
[ØØØ]    21:
[8L7]    22: ;als COM-Programm Platz für den PSP lassen
[AMA]    23:          org 1ØØhassume cs:programm,ds:pro
         gramm,ss:programm,es:programm
[ØØØ]    24:
[DY5]    25: ;Alle Segmente beziehen sich auf das
[3Ø3]    26: ;Programm-Segment
[ØØØ]    27:
[VW5]    28: start     proc far        ;hier beginnt das
[KK2]    29:                           ;Hauptprogramm
[ØØØ]    3Ø:
[V15]    31: ;hier gehört Ihr Programm hin
[ØØØ]    32:
[332]    33:          mov   ax,4cØØh
[L26]    34:    ;beende das Programm über Funktion
[5D5]    35:    ;4Ch ohne Fehlercode über das DOS
[N81]    36:          int   21h
[N35]    37: start     endp       ;hier endet das
[KK2]    38:                      ;Hauptprogramm
[P3Ø]    39: ;
[5U4]    4Ø: ;----------------------------------------
[P3Ø]    41: ;
[3S6]    42: ;Hier können die Daten untergebracht wer-
[ØE6]    43: ;den. Sie befinden sich dann im gleichen
[E25]    44: ;Segment wie der Programm-Code
[P3Ø]    45: ;
[ØØØ]    46:
[DS4]    47: ;hier gehören Ihre Daten hin
[P3Ø]    48: ;
[ØØØ]    49:
[4C2]    5Ø: programm  ends
[X14]    51:         ;hier endet das Segment
[ES1]    52:           end  start
[WF5]    53:         ;hier endet das Listing, es soll
[ØT4]    54:         ;bei START gestartet werden
```

Listing 1. »commaske.asm« dient als Gerüst für Ihre eigenen Com-Programme

```
[P3Ø]     1: ;
[6V4]     2: ; Programmname: EXEMASKE.ASM
[P3Ø]     3: ;
[T16]     4: ; Gerüst für Assembler-Programme,
[4Z5]     5: ; die als EXE-Programme assembliert
[9N2]     6: ; werden sollen
[P3Ø]     7: ;
[WY3]     8: ; (C) 1989 Everts & Hagedorn
[8U3]     9: ;          DOS International
[934]    1Ø: ; Autor: Martin Althaus
[K84]    11: ; letzte Änderung: 12.4.89
[P3Ø]    12: ;
[5U4]    13: ;----------------------------------------
[P3Ø]    14: ;
[6N6]    15: ; In diesem Segment wird nach Aufruf des
[X56]    16: ; Programms der Stapel untergebracht
[P3Ø]    17: ;
[HC4]    18: stapel    segment para stack
[WN3]    19:         ;Segment soll an einem
[CL3]    2Ø:         ;Paragraphen beginnen
[682]    21:           dw 1ØØh dup (?)
[NK5]    22:    ;512 Bytes reservieren, das reicht
[PH2]    23:    ;für 256 Words
[FA5]    24: stapel    ends      ;Segment beenden
[P3Ø]    25: ;
[5U4]    26: ;----------------------------------------
[P3Ø]    27: ;
[EW5]    28: ; In diesem Segment werden die Daten
[NA6]    29: ; untergebracht, zwischen den Segment
[MZ5]    3Ø: ; grenzen können Sie alle Ihre Daten,
[NH7]    31: ; Konstanten, Variablen und Speicherblöcke
[PG1]    32: ; anlegen.
[P3Ø]    33: ;
[5A4]    34: daten     segment para 'data'
[PS3]    35:         ; Segment soll an einem
[FN3]    36:         ; Paragraphen beginnen
```

```
[P3Ø]    37: ;
[VX1]    38: ;hier Daten
[P3Ø]    39: ;
[ØØØ]    4Ø:
[K75]    41: daten     ends       ;Segment beenden
[P3Ø]    42: ;
[5U4]    43: ;----------------------------------------
[P3Ø]    44: ;
[SY6]    45: ; In diesem Segment wird der Programmcode
[5K7]    46: ; untergebracht. Alle Programme, Prozeduren
[N87]    47: ; und Unterprogramme können Sie in dieses
[BU2]    48: ; Segment setzen.
[P3Ø]    49: ;
[PN4]    5Ø: programm  segment para 'code'
[WN3]    51:         ;Segment soll an einem
[CL3]    52:         ;Paragraphen beginnen
[ØØØ]    53:
[WR6]    54:           assume cs:programm, ds:daten, ss:
         stapel
[P3Ø]    55:         ;
[9V3]    56:         ;Zeige dem Assembler an,
[Y44]    57:         ;welches Segment sich auf
[984]    58:         ;welches Segmentregister
[ZB1]    59:         ;bezieht
[ØØØ]    6Ø:
[VK5]    61: start     proc far   ;hier beginnt das
[KK2]    62:                      ;Hauptprogramm
[ØØØ]    63:
[B72]    64:           mov   ax,daten
[SU3]    65:         ;lade Adresse des Daten-
[6C4]    66:         ;segments und übertrage
[WW1]    67:         ;sie nach DS
[XV1]    68:           mov   ds,ax
[P3Ø]    69: ;
[V15]    7Ø: ;hier gehört Ihr Programm hin
[P3Ø]    71: ;
[ØØØ]    72:
[332]    73:           mov   ax,4cØØh
[8F4]    74:         ;beende das Programm über
[YØ4]    75:         ;Funktion 4Ch ohne Fehler
[LY1]    76:         ;code über
[M91]    77:         ;das DOS
[N81]    78:           int   21h
[ØØØ]    79:
[UØ2]    8Ø: start     endp
[G15]    81:         ;hier endet das Hauptprogramm
[ØØØ]    82:
[4C2]    83: programm  ends
[X14]    84:         ;hier endet das Segment
[ES1]    85:           end  start
[K34]    86:         ;hier endet das Listing,
[GV4]    87:         ;es soll bei START gestartet
[GB1]    88:         ;werden
```

Listing 2. »exemaske.asm« können Sie als Gerüst für Ihre eigenen Exe-Programme verwenden

```
[M13]     1: program exetocom;
[ØØØ]     2:
[X34]     3: (* Programmname: exetocom
[ZX5]     4:  Konvertiert EXE- in COM-Programme
[ØØØ]     5:
[7Y5]     6:  (C) Redaktionsbüro Everts & Hagedorn
[833]     7:      DOS International
[VG3]     8:  Autor: Martin Althaus
[BS3]     9:  letzte Änderung: 8.5.89
[F6Ø]    1Ø: *)
[ØØØ]    11:
[RNØ]    12: var
[8F3]    13:  qname,zname:string;
[NR3]    14:  var fq,fz:file of byte;
[212]    15:  qwert: byte;
[RK2]    16:  index: integer;
[7ØB]    17:  parada,dummy,reloz_anzahl,header_groesse,s
         tart_adresse: word;
[ØØØ]    18:
[2R4]    19: (* ---------------------------------------
[FM1]    2Ø:  Funktion:
[VK6]    21:  Wandelt String in Großbuchstaben um
[PK4]    22:  ---------------------------------------- *)
[ØØØ]    23:
[MD6]    24: function hilet(wort: string):string;
[RNØ]    25:  var
[EZ1]    26:    index: byte;
[162]    27:    help:string;
[MUØ]    28:  begin
[122]    29:    help:=wort;
[ZB5]    3Ø:    for index:=1 to length(wort) do
[P57]    31:     if (wort[index]>='a') and (wort[index]<=
         'z')
[1T7]    32:      then help[index]:=char(byte(wort[index])
         -32);
[V22]    33:    hilet:=help;
[ZNØ]    34:  end;
[ØØØ]    35:
```

```
[2R4]   36: (* ----------------------------------------
[W76]   37:  Funktion: gibt eine Fehlermeldung aus
[PK4]   38:  ---------------------------------------- *)
[ØØØ]   39:
[7D5]   4Ø: procedure fehler(nummer: byte);
[6XØ]   41:  const
[HT6]   42:   fehlermeldung: array[1..3] of string[8Ø]
        =
[YV4]   43:     ('Quelldatei nicht gefunden ',
[RZ5]   44:      'Diskette oder Festplatte ist voll ',
[LYC]   45:      'Datei nicht konvertierbar - segmentübe
        rgreifend oder falsche Startadresse');
[MUØ]   46:  begin
[XF1]   47:   writeln;
[4R5]   48:   writeln(fehlermeldung[nummer]);
[XF1]   49:   writeln;
[ZNØ]   5Ø:  end;
[ØØØ]   51:
[2R4]   52: (* ----------------------------------------
[FM1]   53:  Funktion:
[WY6]   54:  Liest zwei Bytes aus einer Datei und macht
[LL2]   55:  daraus ein Wort
[PK4]   56:  ---------------------------------------- *)
[ØØØ]   57:
[D64]   58: function get_word: word;
[RNØ]   59:  var
[5Z3]   6Ø:   qwertlo,qwerthi: byte;
[MUØ]   61:  begin
[MX5]   62:   read(fq,qwertlo); read(fq,qwerthi);
[H85]   63:   get_word:=256*qwerthi+qwertlo;
[ZNØ]   64:  end;
[ØØØ]   65:
[1P1]   66: (* -------------
[NC2]   67:  Hauptprogramm
[8N1]   68:  ------------ *)
[ØØØ]   69:
[MUØ]   7Ø: begin
[XF1]   71:  writeln;
[ØØØ]   72:
[HY5]   73: (* ---- Einschaltmeldung ausgeben ---- *)
[ØØØ]   74:
[2R6]   75:  writeln('-------------------------------
        -----------');
[LZ8]   76:  writeln('--  Konvertierungsprogramm: EXE n
        ach BIN  --');
[ØG8]   77:  writeln('--  entwickelt 5/89 von:  Martin
        Althaus  --');
[ZD9]   78:  writeln('-- für: Redaktionsbüro Everts u.
        Hagedorn --');
[2R6]   79:  writeln('-------------------------------
        -----------');
[XF1]   8Ø:  writeln;
[ØØØ]   81:
[TT5]   82: (* ---- Kommandozeile auswerten ---- *)
[ØØØ]   83:
[TM2]   84:  if paramcount>=1
[2RØ]   85:   then
[HØ3]   86:    qname:=paramstr(1)
[ZRØ]   87:   else
[MUØ]   88:    begin
[ØØØ]   89:
[LM6]   9Ø: (* -------- Solange nach Quelldatei fragen,
[ZY4]   91:              bis eine eingegeben wird ---- *
        )
[ØØØ]   92:
[P31]   93:     repeat
[935]   94:      write('Name der Quelldatei:   ');
[982]   95:      readln(qname);
[SH2]   96:     until qname<>'';
[ZNØ]   97:    end;
[ØØØ]   98:
[MR6]   99: (* ----- An Quelldatei .EXE anhängen, falls
[2Ø5]  1ØØ:          noch nicht mit eingegeben ----- *)
[ØØØ]  1Ø1:
[PT6]  1Ø2:  if pos('.',qname)=Ø then qname:=qname+'.ex
        e';
[ØØØ]  1Ø3:
[TH2]  1Ø4:  if paramcount=2
[2RØ]  1Ø5:   then
[513]  1Ø6:    zname:=paramstr(2)
[ZRØ]  1Ø7:   else
[MUØ]  1Ø8:    begin
[EZ4]  1Ø9:     write('Name der Zieldatei:    ');
[ØØØ]  11Ø:
[626]  111: (* --- Namen der Zieldatei einlesen --- *)
[ØØØ]  112:
[K82]  113:     readln(zname);
[ØØØ]  114:
[8F6]  115: (* ------ falls keine Name eingegeben wird,
[DU4]  116:           benutze Namen der Quelldatei
[7E4]  117:           mit Endung .COM -------------- *)
[ØØØ]  118:
[ZNØ]  119:    end;
[3D2]  12Ø:  if zname='' then
[LC6]  121:   zname:=copy(qname,1,pos('.',qname))+'com'
[ZRØ]  122:  else
[DV6]  123:   if pos('.',zname)=Ø then zname:=zname+'.c
        om';
[ØØØ]  124:
[2Y5]  125: (* ---------- Öffne Quelldatei --------- *)
[ØØØ]  126:
[2U2]  127:  assign(fq,qname);
[MC3]  128:  (*$i-*)  reset(fq); (*$i+*)
[ØØØ]  129:
[GM5]  13Ø: (*   -------- Teste auf Fehler --------- *)
[ØØØ]  131:
[FØ3]  132:  if ioresult=Ø then
[MUØ]  133:  begin
[ØØØ]  134:
[NW5]  135: (* ---------- Öffne Zieldatei ---------- *)
[ØØØ]  136:
[SV2]  137:   assign(fz,zname);
[432]  138:   rewrite(fz);
[ØØØ]  139:
[RN5]  14Ø: (* ---------- Teste auf Fehler --------- *)
[ØØØ]  141:
[FØ3]  142:   if ioresult=Ø then
[MUØ]  143:    begin
[ØØØ]  144:
[X95]  145:   (* ------- Lies den Header aus ------- *)
[ØØØ]  146:
[K86]  147:     for index:=1 to 3 do dummy:=get_word;
[GA4]  148:     reloz_anzahl:=get_word;
[5H4]  149:     header_groesse:=get_word;
[P86]  15Ø:     for index:=1 to 5 do dummy:=get_word;
[2E4]  151:     start_adresse:=get_word;
[K86]  152:     for index:=1 to 3 do dummy:=get_word;
[ØØØ]  153:
[8Ø6]  154:   (* ---Ist die Datei konvertierbar? --- *)
[ØØØ]  155:
[RY7]  156:     if (reloz_anzahl=Ø) and (start_adresse=
        256) then
[MUØ]  157:      begin
[ØØØ]  158:
[R85]  159:    (* ------ ja, konvertiere sie ------- *)
[ØØØ]  16Ø:
[XF1]  161:       writeln;
[SV9]  162:       writeln('Ich konvertiere: ',hilet(qna
        me),' => ',hilet(zname));
[ØØØ]  163:
[8G4]  164:      (* ------ lies nicht benötigte
[VH3]  165:                Bytes aus --------------- *)
[ØØØ]  166:
[RN9]  167:       for index:=1 to header_groesse*16-28+
        256 do read(fq,qwert);
[P31]  168:        repeat
[ØØØ]  169:
[ØE5]  17Ø:         (* ---------- Übertrage die einzel-
[DV2]  171:                    nen Bytes -------- *)
[ØØØ]  172:
[9L2]  173:         read(fq,qwert);
[VR2]  174:         write(fz,qwert);
[S72]  175:        until eof(fq);
[BX8]  176:       writeln('Konvertierung erfolgreich du
        rchgeführt.');
[4MØ]  177:      end
[ZRØ]  178:     else
[ØØØ]  179:
[9Z3]  18Ø:     (* ------ Fehler: Datei ist
[9C4]  181:               nicht konvertierbar ------ *)
[ØØØ]  182:
[XG1]  183:       fehler(3);
[ØØØ]  184:
[HC5]  185:     (* ----- Schließe die Dateien ------ *)
[ØØØ]  186:
[7L1]  187:     close(fq);
[GL1]  188:     close(fz);
[4MØ]  189:    end
[ZRØ]  19Ø:   else
[MUØ]  191:    begin
[ØØØ]  192:
[D82]  193:    (* ------- Fehler:
[8N4]  194:               Datenträger ist voll ----- *)
[ØØØ]  195:
[WG1]  196:     fehler(2);
[GL1]  197:     close(fz);
[4MØ]  198:    end
[4MØ]  199:  end
[ZRØ]  2ØØ:   else
[ØØØ]  2Ø1:
[252]  2Ø2:   (* ------ Fehler:
[1M4]  2Ø3:             Datei nicht gefunden ------- *)
[ØØØ]  2Ø4:
[VG1]  2Ø5:    fehler(1);
[XF1]  2Ø6:  writeln;
[KNØ]  2Ø7:  end.
```

Listing 3. »exe2com.pas« ist eine Spur komfortabler als sein Vorbild – ansonsten gibt es keine Unterschiede

Makroassembler SAM III

Der DOS-Assembler zum Nulltarif

Nachdem der mittlerweile knapp zwei Jahre alte Makroassembler SAM in der DOS Extra 1 seinen Einstand feierte und in der DOS 5'88 im Rahmen eines Assemblerkurses zur Version II fortentwickelt wurde, liegt nun die langerwartete Version III vor, die allen bislang geäußerten Wünschen der Anwender Rechnung trägt und sich mit den Compilern SBasic, Quick Basic und Turbo Basic übersetzen läßt.

Für PC-Besitzer ist die Auswahl an Assemblern immer noch sehr spärlich. Am bekanntesten sind die Makroassembler MASM von Microsoft und TASM von Borland. Beide Assembler sind sehr gut, leider aber auch mehrere hundert Mark teuer. Eine preiswerte und trotzdem leistungsfähige Alternative bietet der DOS-Assembler SAM, der nun in der dritten Version vorliegt (Listing 1). SAM III ist ein Makroassembler, der Programme mit mehreren Segmenten verarbeiten kann, Prozeduren unterstützt und Exe-Dateien erzeugt, die sich auch in Com-Dateien umwandeln lassen – den dazu erforderlichen Aufbau vorausgesetzt.

Um die Größe von SAM III nicht ausufern zu lassen, wurden die Daten zum Übersetzen der Assemblerprogramme in eine eigene Datei namens »sam3.dat« (Listing 2) ausgelagert. Diese Datei muß sich immer im gleichen Verzeichnis wie SAM III selbst befinden.

SAM III ist der Makroassembler für den kleinen Geldbeutel

Um eine lauffähige Version von SAM III zu erhalten, empfiehlt es sich, wie folgt vorzugehen: Geben Sie zuerst »sam3.dat« mit Hilfe des Checksummers »SCheck« (das erste Listing in dieser DOS-Extra-Ausgabe) ein, und speichern Sie die Datei unter dem angegebenen Namen.

Daraufhin tippen Sie den Quelltext »sam3.asc« von SAM III (ebenfalls mit Unterstützung von SCheck) ein. Es ist Ihnen sicherlich schon aufgefallen, daß »sam3.asc« keine Zeilennummern enthält. Der Vorteil des strukturierten Programmaufbaus und der Verwendung von Sprungmarken (Labels) anstelle von Zeilennummern ist die verbesserte Lesbarkeit des Programms, die das Verständnis und die Wartung erheblich erleichtert.

Der Nachteil dieser Form besteht darin, daß das Programm mit GW-Basic nicht läuft. Doch keine Sorge, dafür läßt sich das Listing einwandfrei mit SBasic (aus DOS Extra 6), Quick Basic und Turbo Basic compilieren, und gleichzeitig erhalten Sie eine Version mit einer Ausführungsgeschwindigkeit, die GW-Basic vor Neid erblassen ließe.

SAM III läßt sich wahlweise mit SBasic, Quick Basic oder Turbo Basic compilieren

Wenn Sie SAM III mit SBasic compilieren wollen, benötigen Sie zusätzlich den Präprozessor aus DOS Extra 7, der das Listing in ein GW-Basic-kompatibles Format umwandelt. Natürlich könnten Sie dann auch SAM III unter GW-Basic benutzen (Aufruf mit »basic sam3 /f:6«), doch ist das Arbeiten bei der langsamen Ausführungsgeschwindigkeit alles andere als bequem. Die SBasic-Version arbeitet da schon etwa neunmal so schnell, so daß Sie die zehn Minuten, die das Übersetzen von »sam3.asc« in »sam3.exe« mit SBasic dauert, ruhig investieren sollten.

Wenn Sie SAM III mit Quick oder Turbo Basic compilieren möchten, müssen Sie den Teil am Anfang von Listing 1 zwischen dem Kommentar

```
»Überprüfen, ob in der«
```

und dem ersten Auftreten des Kommentars

```
»Initialisierung«
```

weglassen. Das bedeutet allerdings auch, daß die Übergabe von Parametern in der Quick- und Turbo-Basic-Version entfällt. Dieses Manko können Sie nur ausgleichen, indem Sie auf SBasic zurückgreifen.

Unter Quick Basic 3.0 compilieren Sie SAM III am schnellsten aus der MS-DOS-Kommandoebene heraus mit der Befehlsfolge

```
qb sam3.asc /o /q /s ;
```

Benutzen Sie Quick Basic 4.0 oder 4.5, so schreiben Sie statt dessen

```
bc sam3.asc /o /s ;
```

In beiden Fällen müssen Sie abschließend das Zwischenergebnis in Form der Objektdatei »sam3.obj« mit dem MS-DOS-Linker durch

```
link sam3 ;
```

zur lauffähigen SAM-Version binden.

Um SAM III mit Turbo Basic zu compilieren, sind keine besonderen Einstellungen nötig. Allerdings ist in SAM III das Format bei der Ausgabe eines assemblierten Programms etwas wirr, wenn Sie den Assembler mit Turbo Basic übersetzen. Dem korrekten Funktionieren des Assemblers tut dieser Umstand jedoch keinen Abbruch.

Die Quick- und die Turbo-Basic-Version von SAM III haben den Vorteil,

```
SAM III - Der Makro-Assembler der DOS International - (C) 1989 DOS International
------------------------------------------------------------------------------

                              S A M  III

                        Geschrieben von Markus Zietlow

------------------------------------------------------------------------------
SAM-Datei:makro.SAM
EXE-Datei:makro.EXE
------------------------------------------------------------------------------
<1> Quelltextausgabe auf Bildschirm      <2> Quelltextausgabe auf einem Drucker
<3> Generierung einer EXE-Datei          <4> Verzeichnis anzeigen
<5> Assemblerlisting  (auf Bildschirm)   <6> Ohne Funktion (reserviert)
<7> Zweiter Include-Lauf ? (Nein)        <8> Beenden von SAM III
<9> Eingabe des Namens der SAM-Datei     <0> Eingabe des Namens der EXE-Datei
```

Bild 1. Die Benutzeroberfläche des Makro-Assemblers SAM III

daß sie gegenüber SBasic um beinahe 80 Prozent schneller sind, doch dafür ist der Programmcode der SBasic-Version auch nur halb so groß wie der Code der Quick-Basic- und zweidrittel so groß wie der Code der Turbo-Basic-Version. Generell arbeitet die SBasic-Version erheblich speicherplatzsparender.
SAM III läßt sich mit einer Benutzeroberfläche oder beim Einsatz von SBasic auch kommandozeilenorientiert aufrufen. Um in die Benutzeroberfläche zu gelangen, starten Sie SAM III mit

```
sam3
```

und es erscheinen das Menü und Titelbild von SAM III (Bild 1).
Möchten Sie dagegen unter SBasic kommandozeilenorientiert arbeiten, so rufen Sie SAM III mit

```
samii quelldat zieldat
```

auf, und SAM III assembliert ohne weitere Eingabe selbständig die Datei »quelldat.sam« zur Datei »zieldat.exe« und kehrt in die MS-DOS-Kommandoebene zurück (Bild 2).
Wenn Sie aus Geschwindigkeitsgründen beim Assemblieren einer Datei auf die Bildschirmausgabe des Quelltextes und seiner Übersetzung in hexadezimaler Form verzichten möchten, müssen Sie beim Aufruf von SAM III den Parameter » – « (ein Minuszeichen) mit angeben.

SAM III arbeitet in der SBasic-Version auf Wunsch auch kommandozeilenorientiert

Falls die Datei »quelldat« ohne explizit aufgeführte Speichersegmente arbeitet (beispielsweise das Programm in Listing 3), können Sie die Exe-Datei mit dem Befehl

```
exe2bin name.exe name.com
```

auch in eine Com-Datei umwandeln. Voraussetzung ist natürlich, daß Sie das MS-DOS-Hilfsprogramm »ex2bin« besitzen. Ist dies nicht der Fall, dann verwenden Sie einfach das dazu kompatible Programm »exe2com« aus dieser DOS-Extra-Ausgabe.
Die Stapeldatei »sam.bat« in Listing 4 nimmt Ihnen die lästige Arbeit des Tippens der verschiedenen Dateinamen ab. Sie wird mit dem Befehl

```
sam3 quelldat
```

aufgerufen und übersetzt den Quelltext »quelldat.sam« automatisch in eine Com-Datei namens »quelldat.com«.
Wenn Sie die Erweiterungen der einzelnen Dateinamen nicht angeben, nimmt SAM III automatisch an, daß es sich um den Typ »sam« beziehungsweise »exe« handelt.
Im Unterschied zu SAM II funktioniert der kommandozeilenorientierte Aufruf von SAM III unter SBasic jetzt auch bei der Arbeit mit speicherresidenten Programmen.
Die Assemblerprogramme für SAM III lassen sich mit jeder Textverarbeitung schreiben, die reine ASCII-Dateien erzeugt (zum Beispiel Word oder PC-Write). Im Zweifelsfall tut es auch der MS-DOS-Zeileneditor »edlin.com«. Achten Sie aber darauf, daß Sprungmarken und Variablen in der ersten Spalte beginnen.
Um Programme, die größer als 64 KByte sind, erzeugen zu können, verarbeitet SAM III, wie die meisten anderen Assembler auch, Segmentierungen. Listing 5 zeigt dazu ein Beispielprogramm, das die drei ersten Buchstaben des Alphabets auf dem Bildschirm ausgibt.
Eine interessante Erweiterung bei SAM III stellt der Befehl »include« dar. Dieser Befehl fügt den Quelltext mit dem angegebenen Dateinamen in das aktuelle Listing ein. Listing 6 zeigt ein Beispiel für das Einbinden einer Include-Datei, und Listing 7 die Include-Datei selbst.

Die Pseudobefehle von SAM III

Befehl	Bedeutung
Symboldefinitionen:	
equ	Weist einem symbolischen Namen einen konstanten Wert zu `Beispiel: anzahl equ 255`
=	Entspricht dem Pseudobefehl »equ« `Beispiel: wert = 255`
Datendefinitionen:	
db	Definiert einen Bytespeicherbereich für Variablen oder Konstanten. Es lassen sich auch mehrere Werte zuweisen. `Beispiel: text db 'Dies ist ein Text',10,13,'$'`
ds	Reserviert einen Speicherbereich. Der erste Parameter gibt die Länge, der zweite Parameter den Inhalt des Puffers an. `Beispiel: puffer ds 100h,0`
Makrodefinitionen:	
macro	Die vorausgehende Marke (Label) definiert ein Makro mit den nachfolgenden Parametern (durch Prozentzeichen mit Ziffern repräsentiert) `Beispiel: ausgabe macro %1,%2,%3`
endm	Ende einer Makrodefinition. Ein Makroname wird nicht benötigt. `Beispiel: endm`
do	Aufruf eines zuvor definierten Makros, der folgende Parameter gibt den Makronamen an, die weiteren Parameter werden in das Makro übernommen `Beispiel: do ausgabe,al,20h,dx`
Segmentbefehle:	
segment	Kennzeichnet den Beginn eines Segments `Beispiel: prog segment code`
ends	Markiert das Ende eines Segments `Beispiel: prog ends`
Weitere Befehle:	
org	Definiert die Startadresse des Programms innerhalb des Segments `Beispiel: org 100h`
proc	Kennzeichnet den Beginn einer Prozedur (Unterprogramm) `Beispiel: ausgabe proc`
endp	Markiert das Ende einer Prozedur. Ein Prozedurname wird nicht benötigt. `Beispiel: endp`
end	Gibt das Ende des Assemblerprogramms an
even	Setzt den Befehlszähler auf eine gerade Adresse
include	Fügt den angegebenen Quelltext in das Listing ein. Achten Sie darauf, daß das eingefügte Listing fehlerfrei arbeitet.
Ausgabesteuerung:	
title	Definiert den Titel für ein Listing (wird ignoriert)
subttl	Definiert den Untertitel für ein Listing (wird ignoriert)

Tabelle. Alle Pseudobefehle von SAM III auf einen Blick

```
Adr. Hex-Code    Zeile  Quelltext

0000                1   PROG    SEGMENT CODE  ;Codesegmentanfang
0000                2   ;
0000 B402           3           MOV AH,2      ;Die Buchstaben
0002 B241           4           MOV DL,65     ; 'A'
0004 CD21           5           INT 21H       ; und
0006 B242           6           MOV DL,66     ; 'B'
0008 CD21           7           INT 21H       ; und
000A B243           8           MOV DL,67     ; 'C'
000C CD21           9           INT 21H       ;nacheinander auf
000E B44C           10           MOV AH,4CH    ;den Bildschirm
0010 CD21           11           INT 21H       ;ausgeben
0012                12  ;
0012                13  PROG    ENDS          ;Codesegmentende
0012                14  ;
0012                15  STAPEL SEGMENT STACK ;Stapelsegmentanfang
0012 00000000       16          DB  0,0,0,0
0016                17  STAPEL ENDS          ;Stapelsegmentende
0016                18  ;
0016                19          END           ;Programmende

Durchlauf 4 ist aktiv.                  Anzahl der bearbeiteten Zeilen:  18
```

Bild 2. SAM III beim Assemblieren einer Sam-Datei

Zur Leistungsfähigkeit von SAM III trägt entscheidend die Fähigkeit zur Makroprogrammierung bei. Sie können bei Makroaufrufen und -definitionen bis zu acht Parameter übergeben beziehungsweise vereinbaren, die von dem Assembler problemlos verarbeitet werden.

Die Fähigkeit zur Makroassemblierung trägt entscheident zur Leistungsfähigkeit von SAM III bei

Die Makrodefinition beginnt mit der Angabe des Makronamens, der vom Befehl »macro« und den Parametern gefolgt ist. Die Zeile

```
drucken macro %1,%2,%3
```

definiert beispielsweise das Makro »drucken«, an das drei Parameter übergeben werden. Daraufhin müssen die Programmzeilen folgen, die parameterabhängig ausgeführt werden sollen, wie zum Beispiel

```
mov ah,%1
mov al,%2
mov cx,%3
```

Beendet wird eine Makrodefinition durch den Befehl »endm«.
Um ein Makro in den Programmtext einzufügen, müssen Sie es mit dem Do-Befehl aufrufen. Wenn beispielsweise das Makro »drucken« mit

```
do drucken,AL,20h,DX
```

angesprochen wird, so fügt SAM III die Befehle des Makros automatisch in den Programmtext ein und ersetzt dabei »%1« durch »AL«, »%2« durch »20h« und »%3« durch »DX«. Wird das Makro hingegen mit

```
do drucken,,,32
```

aktiviert, so ersetzt SAM III ausschließlich »%3« durch »32«. Die Programmzeilen, in denen »%1« und »%2« enthalten sind, werden gelöscht.
Listing 8 demonstriert die praktische Anwendung der Makrobefehle. Weil das Makrobeispiel mit nur einem Segment, dem Codesegment, arbeitet, läßt es sich nach der Assemblierung mit »exe2bin« in eine Com-Datei umwandeln.
Um den Sinn der Makroprogrammierung zu verstehen, betrachten Sie Listing 9. Dieses Programm arbeitet wieder mit dem Speichermodell für Com-Dateien. Die Befehlsfolge

```
mov dx,offset(adresse)
mov ah,nummer
int 21h
```

zum Aufruf einer Funktion des Interrupt 21hex kommt in Listing 9 mehrfach vor. Der einzige Unterschied zwischen den Aufrufen besteht in den Werten der angesprochenen Adressen.
In langen Assemblerprogrammen kommt diese Befehlsfolge noch weitaus häufiger vor, doch muß Sie deshalb nicht jedesmal neu eingegeben werden. Zu diesem Zweck ist SAM III nämlich makrofähig. Makros sind immer wiederkehrende Befehlsfolgen, die der Programmierer bei der Arbeit mit einem Assembler als Makro definieren kann und dadurch nur ein einziges Mal schreiben muß. Wenn er die in einem Makro enthaltenen Befehle an einer bestimmten Stelle in seinem Programm benötigt, setzt er lediglich den Namen des Makros ein. Der Assembler fügt dann beim Übersetzen des Programms in Maschinencode den Inhalt des Makros automatisch an den entsprechenden Stellen ein. Die Makroversion des Beispiels in Listing 9 finden Sie in Listing 10.
Anstelle von Makros können Sie auch Unterprogramme verwenden. Doch im Gegensatz zu einem Makro muß das Unterprogramm jedesmal zeitraubend über den Befehl »call« aufgerufen und mit dem Befehl »ret« beendet werden, während das Makro direkt an der gewünschten Stelle im Programm vorhanden ist und die Geschwindigkeit nicht herabsetzt.
Unterprogramme besitzen jedoch auch einen nicht zu verachtenden Vorteil: Sie sind nämlich nur ein einziges Mal im Programm vorhanden, während der Assembler das Makro an jeder Stelle, an der es benötigt wird, neu einsetzt.
Mit SAM III lassen sich wie auch beim MASM Prozeduren mit den beiden Befehlen »proc« und »enp« definieren. Sie dürfen dabei die Prozeduren beliebig verschachteln. Ein Beispiel für ein Programm mit Prozeduren sehen Sie in Listing 11.

Neben Makros erlaubt SAM III auch die Verwendung von Prozeduren und Include-Dateien

Abschließend einige Hinweise zu den Unterschieden zwischen MASM- und SAM-III-Assemblerprogrammen: Weil der Parser (der Programmteil, der das Assemblerprogramm in Marken, Befehle, Operanden und Kommentare zerlegt) von SAM III einfach (aber schnell) arbeitet, müssen Sie bei der Adressierung über einen Offset-Befehl die Marke, deren Adresse ermittelt werden soll, in runde Klammern einschließen. Zwischen »offset« und der Marke darf kein Leerraum stehen, sonst erkennt der Parser die Marke aufgrund des Leerzeichens nämlich nicht. Beim Zerlegen der Programmzeile würde der Rest einfach abgeschnitten. Ein Beispiel:

```
mov ax,offset(text)
```

Aus dem gleichen Grund muß das Displacement (die Verschiebung) bei der indirekten Adressierung ohne Pluszeichen direkt vor die eckige Klammer, die die Quell- oder Zieladresse enthält, geschrieben werden. Einige Beispiele dazu:

```
mov ax,80h[BX+SI]
mov cx,index[DX]
mov -16[di],si
```

SAM unterscheidet übrigens nicht zwischen Groß- und Kleinschreibung, so daß die Schreibweise Ihnen überlassen bleibt.
Der Aufbau einer Assemblerprogrammzeile für SAM entspricht dem üblichen Standard. Jede Zeile besteht aus maximal vier Feldern, die von links beginnend Marken und Variablen, Befehle, Operanden und Kommentare enthalten. Die linke Spalte ist nur für Marken und Variablen vorgesehen. Haben Sie keine Angabe für

die erste Spalte, so müssen Sie den Befehl in dieser Zeile mit mindestens einem Leerzeichen oder einem Tabulatorzeichen einleiten.

Die Syntax bei SAM III weicht in einigen Fällen von der beim MASM und TASM verwendeten Syntax ab

Variablen und Marken werden bei SAM III ohne Doppelpunkt angegeben. Für Marken ist jedoch der Doppelpunkt zulässig. Kommentartexte müssen Sie – wie allgemein üblich – mit einem Semikolon anführen, und in jeder Programmzeile muß der Kommentar von neuem durch ein Semikolon gekennzeichnet werden. Texte dürfen nur in einfache Hochkommata (»'«) eingeschlossen werden, das doppelte Anführungszeichen »"« ist nicht zulässig.
Die Sprungbefehle »jmp« und »call« müssen bei SAM III direkt von einem »n« oder »f« gefolgt sein, wenn die Zieladresse zu den Befehlen in einem Register angegeben ist. Das »n« (für »near«) steht für einen kurzen Sprung innerhalb des Segments (Intrasegment), und das »f« (für »far«) gibt entsprechend einen weiten Sprung über die Segmentgrenzen hinweg (Intersegment) an. Zwei Beispiele:

```
jmps [SI]
calln [DI]
```

Möchten Sie Daten über die Programmgrenzen hinaus schreiben oder lesen (beispielsweise in den Bildschirmspeicher), so müssen Sie das Segmentregister DS mit der gewünschten Adresse laden. Wollen Sie beispielsweise den Wert des ersten Byte des Bildschirmspeichers einer CGA-Farbgrafikkarte in das AL-Register lesen, so müssen Sie

```
mov ax,B800h ;Segmentadresse
mov ds,ax ;in DS schreiben
mov si,0 ;und erste Speicher-mov al,[si]
;zelle auslesen
```

schreiben.
Auch eine Verzweigung über die Segmentgrenzen hinweg, beispielsweise in das BIOS, ist erlaubt:

```
basis equ F000h
adresse equ 1000h
jmp adresse,basis
```

Ein entsprechender Unterprogrammaufruf sieht folgendermaßen aus:

```
bios equ F000h
adresse equ 2000h
mov [di],adresse
mov 2[di],bios
callf [di]
```

Die jeweils angegebenen BIOS-Adressen sind natürlich rein fiktiv und von der BIOS-Version Ihres Computers abhängig.

Sprungbefehle mit den Zieladressen in Registern benötigen den Zusatz »n« oder »j«

Eine weitere Besonderheit von SAM III: Die Präfixbefehle »lock«, »rep« und »seg«, die normalerweise direkt vor den betreffenden Befehl gehören, sind bei SAM III in eine eigene Zeile zu schreiben, damit sie korrekt verarbeitet werden.
Auch die Pseudobefehle von SAM III sind bemerkenswert. Die Tabelle zeigt eine Aufstellung aller Pseudobefehle von SAM III im Überblick. Der Pseudobefehl »ds« ist gegenüber dem MASM und TASM neu und reserviert einen Speicherbereich, beispielsweise für einen Zeichenpuffer.

(Markus Zietlow/ma)

```
[AZ3]     1: 'Makro-Assembler SAM III
[4W6]     2: '(C) 1989 Redaktionsbüro Everts&Hagedorn
[5P5]     3: '          Redaktion DOS International
[M16]     4: '          Geschrieben von Markus Zietlow
[166]     5: '          Veröffentlicht in DOS Extra 8
[TL1]     6: DEFINT A-Z
[PNØ]     7: CLS
[E2Ø]     8: '
[GA6]     9: 'Überprüfen, ob in der Kommandozeile
[RV4]    1Ø: ' Parameter übergeben wurden
[XA9]    11: quelldat$ = COMMAND$(1) : zieldat$ = COMMAN
              D$(2) : ausgabe$ = " "
[E2Ø]    12: '
[6L5]    13: 'Bearbeiten des Quelltextnamens
[5DC]    14: IF quelldat$ = "-" THEN ausgabe$ = "-": que
              lldat$ = "": zieldat$ = "": GOTO pruefen
[617]    15: IF zieldat$ = "-" THEN ausgabe$ = "-": ziel
              dat$ = ""
[E2Ø]    16: '
[WF1]    17: pruefen:
[HW2]    18: pruef$ = quelldat$
[E2Ø]    19: '
[H57]    2Ø: 'Feststellen, ob Erweiterung angegeben ist
[GE4]    21: WHILE INSTR(pruef$, "\") <> Ø
[UL6]    22:   pruef$ = MID$(pruef$, INSTR(pruef$, "\")
              + 1)
[TRØ]    23: WEND
[E2Ø]    24: '
[HNB]    25: IF (INSTR(pruef$, ".") = Ø) AND (pruef$ <>
              "") THEN quelldat$ = quelldat$ + ".SAM"
[KR4]    26: IF zieldat$ = "" THEN GOTO init
[E2Ø]    27: '
[YG5]    28: 'Bearbeiten des Zieldateinamens
[Ø51]    29: DOS = -1
[ZM2]    3Ø: pruef$ = zieldat$
[E2Ø]    31: '
[H57]    32: 'Feststellen, ob Erweiterung angegeben ist
[GE4]    33: WHILE INSTR(pruef$, "\") <> Ø
[UL6]    34:   pruef$ = MID$(pruef$, INSTR(pruef$, "\")
              + 1)
```

Listing 1. Die dritte SAM-Version »sam3.asc« läßt sich mit SBasic, Quick Basic und Turbo Basic compilieren

Ist Ihr Programm der HIT ?

Der DMV Verlag sucht ständig nach neuer, interessanter Software zur Aufnahme in unser Softwaresortiment.

Dabei ist es einerlei, ob Sie nun ein Anwendungs- oder ein Spielprogramm geschrieben haben.

Der DMV Verlag bietet Ihnen sein Software-Know How an!

Die Inserenten

Bartsch und Kloft....55
Becos....113
BMC....83
Christel's Software....37
CPS Computertechnik....2
Data Norm....76
DMV....20,21,53,123,127,129,155,179
Interest-Verlag....50
Kayser Datentechnik....105
Logibyte....51
Micromarket Worms....113
ROBIN WOOD....99
SHC Computer Fink....76
Star Division....180
T+S GmbH....55
Zoschke, Ing. Büro....113

```
[TR0]  WEND
[E20]  '
[VF8]  IF INSTR(pruef$, ".") = 0 THEN zieldat$ = zieldat$ + ".EXE"
[E20]  '
[LW2]  'Initialisierung
[8V0]  init:
[A26]  'Assemblerlisting auf den Bildschirm
[EH1]  asslis = 1
[2P4]  IF ausgabe$ = "-" THEN asslis = 4
[RB3]  'Keine Symboltabelle
[PK1]  symaus = 0
[FU4]  'Keine Cross-Referenztabelle
[5C1]  cross = 0
[ZT3]  'Datenzeilen pro Seite
[VL1]  ln.max = 72
[K23]  'Zeichen pro Zeile
[EL1]  br.max = 80
[PX2]  wahr = -1: falsch = 0
[GZ2]  trenn$ = " ," + CHR$(9)
[0H2]  makros = falsch
[E20]  '
[BU4]  'Definition der Datenfelder
[LY1]  DIM param$(8)
[6N5]  maxbefehle = 255: stapelgroesse = 10
[2F6]  DIM befehltab(255): DIM prozedurtyp(10)
[E20]  '
[YP4]  'Einlesen der Datei 'SAM3.DAT'
[YT4]  OPEN "sam3.dat" FOR INPUT AS #3
[PY4]  IF EOF(3) THEN GOTO samdat.fehl
[LV5]  INPUT #3, befehlzahl: LINE INPUT #3, b$
[AW7]  DIM kommando$(218), operwert(218), quellregtyp(218)
[4K4]  DIM zielregtyp(218), flag(218)
[YS3]  FOR i = 1 TO befehlzahl
[MZB]    INPUT #3, kommando$(i), operwert(i), zielregtyp(i), quellregtyp(i), flag(i)
[U11]  NEXT i
[E20]  '
[N87]  INPUT #3, vordef, symbolzahl: LINE INPUT #3, b$
[NB8]  DIM symbol$(500), wert1(500), wert2(29), symboltyp(500)
[743]  FOR i = 1 TO vordef
[VC8]    INPUT #3, symbol$(i), wert1(i), wert2(i), symboltyp(i)
[U11]  NEXT i
[343]  aktsymbol = vordef
[251]  CLOSE #3
[BA4]  IF DOS THEN GOTO assembler
[E20]  '
[MG7]  'Titelbild, Ausgabe der Benutzeroberfläche
[094]  ' und Copyright-Meldungen
[E11]  menue:
[PN0]  CLS
[AZD]  PRINT "SAM III - Der Makro-Assembler der DOS International - (C) 1989 DOS International";
[PC3]  PRINT STRING$(80, "-");
[BB5]  LOCATE 16, 1: PRINT STRING$(80, "-");
[1S2]  PRINT "SAM-Datei:"
[9S2]  PRINT "EXE-Datei:"
[PC3]  PRINT STRING$(80, "-");
[C3E]  PRINT "<1> Quelltextausgabe auf Bildschirm        <2> Quelltextausgabe auf einem Drucker"
[D3B]  PRINT "<3> Generierung einer EXE-Datei            <4> Verzeichnis anzeigen"
[98B]  PRINT "<5> Assemblerlisting (        )            <6> Ohne Funktion (reserviert)"
[XSA]  PRINT "<7> Zweiter Include-Lauf ? (Nein)          <8> Beenden von SAM III"
[A1D]  PRINT "<9> Eingabe des Namens der SAM-Datei       <0> Eingabe des Namens der EXE-Datei";
[E20]  '
[Y21]  titel:
[TZ7]  LOCATE 4, 20: PRINT CHR$(201); STRING$(39, 205); CHR$(187)
[ACH]  LOCATE 5, 20: PRINT CHR$(186); " ████  ██   █   █   ███████"; CHR$(186)
[DMC]  LOCATE 6, 20: PRINT CHR$(186); " █  █  ██  ██   █ █ █"; CHR$(186)
[5FG]  LOCATE 7, 20: PRINT CHR$(186); " ████ █ ████ █ ██ █   █ █ █"; CHR$(186)
[T8C]  LOCATE 8, 20: PRINT CHR$(186); " █   █   █   █   █ █ █"; CHR$(186)
[L6H]  LOCATE 9, 20: PRINT CHR$(186); " ████ █   █   █  ███████"; CHR$(186)
[848]  LOCATE 10, 20: PRINT CHR$(200); STRING$(39, 205); CHR$(188)
[VB8]  LOCATE 13, 26: PRINT "Geschrieben von Markus Zietlow"
[E20]  '
[5C1]  status:
[VM9]  LOCATE 17, 11: PRINT quelldat$; STRING$(70 - LEN(quelldat$), 32);
[789]  LOCATE 18, 11: PRINT zieldat$; STRING$(70 - LEN(zieldat$), 32);
[YDV]  LOCATE 22, 24: IF asslis = 1 THEN PRINT "auf Bildschirm" ELSE IF asslis = 2 THEN PRINT "auf Drucker    " ELSE IF asslis = 3 THEN PRINT "in eine Datei " ELSE PRINT "nicht ausgeben"
[E20]  '
[3G6]  'Eingabe des gewünschten Menüpunktes
[UB1]  eingabe:
[RS5]  a$ = INKEY$: IF a$ = "" THEN GOTO eingabe
[T02]  a = ASC(a$) - 48
[VF5]  IF (a < 0) OR (a > 9) THEN GOTO eingabe
[E20]  '
[BE7]  'Ausgabe des Quelltextes auf den Bildschirm
[7L4]  IF a = 1 THEN GOTO aus.monitor
[E20]  '
[UT6]  'Ausgabe des Quelltextes auf den Drucker
[5K4]  IF a = 2 THEN GOTO aus.drucker
[E20]  '
[EZ6]  'Assemblieren des angegebenen Quelltextes
[V74]  IF a = 3 THEN GOTO assembler
[E20]  '
[L56]  'Anzeigen des Inhaltsverzeichnisses
[VG4]  IF a <> 4 THEN GOTO sam.finis
[SL1]  LOCATE 3, 1
[772]  FOR i = 1 TO 13
[R93]    PRINT STRING$(80, 32);
[U11]  NEXT i
[SL1]  LOCATE 3, 1
[YV1]  FILES "*.sam"
[6P5]  PRINT "- Bitte eine Taste drücken -"
[5H3]  WHILE INKEY$ = "": WEND
[SL1]  LOCATE 3, 1
[772]  FOR i = 1 TO 13
[R93]    PRINT STRING$(80, 32);
[U11]  NEXT i
[WT1]  GOTO titel
[E20]  '
[N83]  'Beenden von SAM III
[7S1]  sam.finis:
[G1B]  IF a = 8 THEN CLS : PRINT "SAM III hat seine Arbeit beendet.": PRINT : END
[E20]  '
[ZS4]  'Form des Assembler-Listings
[545]  IF a <> 5 THEN GOTO quell.eingabe
[P23]  asslis = asslis + 1
[2C4]  IF asslis = 5 THEN asslis = 1
[R62]  GOTO status
[E20]  '
[PW4]  'Eingabe des Quelltextnamens
[FB2]  quell.eingabe:
[PZ4]  IF a <> 9 THEN GOTO ziel.eingabe
[SL1]  LOCATE 3, 1
[772]  FOR i = 1 TO 13
[R93]    PRINT STRING$(80, 32);
[U11]  NEXT i
[9HC]  LOCATE 8, 1: LINE INPUT "Geben Sie bitte den Namen der SAM-Datei ein: "; quelldat$
[XL1]  LOCATE 8, 1
[M12]  FOR q = 1 TO 4
[R93]    PRINT STRING$(80, 32);
[K21]  NEXT q
[4X4]  quelldat$ = LEFT$(quelldat$, 66)
[HW2]  pruef$ = quelldat$
[GE4]  WHILE INSTR(pruef$, "\") <> 0
[UL6]    pruef$ = MID$(pruef$, INSTR(pruef$, "\") + 1)
[TR0]  WEND
[TX8]  IF INSTR(pruef$, ".") = 0 THEN quelldat$ = quelldat$ + ".SAM"
[WT1]  GOTO titel
[E20]  '
[5R4]  'Eingabe des Zieldateinamens
[Y42]  ziel.eingabe:
[L64]  IF a <> 0 THEN GOTO in.lauf
[SL1]  LOCATE 3, 1
[772]  FOR i = 1 TO 13
[R93]    PRINT STRING$(80, 32);
[U11]  NEXT i
[Z9C]  LOCATE 8, 1: LINE INPUT "Geben Sie bitte den Namen der EXE-Datei ein: "; zieldat$
[XL1]  LOCATE 8, 1
[M12]  FOR q = 1 TO 4
[R93]    PRINT STRING$(80, 32);
[K21]  NEXT q
[DH4]  zieldat$ = LEFT$(zieldat$, 66)
[ZM2]  pruef$ = zieldat$
[GE4]  WHILE INSTR(pruef$, "\") <> 0
[UL6]    pruef$ = MID$(pruef$, INSTR(pruef$, "\") + 1)
[TR0]  WEND
[VF8]  IF INSTR(pruef$, ".") = 0 THEN zieldat$ = zieldat$ + ".EXE"
[WT1]  GOTO titel
[E20]  '
[D36]  'Zweiter Include-Lauf zum Einbinden
[244]  ' externer Makros nötig ?
[LA1]  in.lauf:
[884]  IF a <> 7 THEN GOTO eingabe
```

```
[EFG]  196: IF makros THEN makros = falsch: LOCATE 23,
            29: PRINT "Nein"; : ELSE makros = wahr: LOC
            ATE 23, 29: PRINT " Ja ";
[T12]  197: GOTO eingabe
[E2Ø]  198: '
[3K5]  199: 'Ausgabe eines Quelltextes auf den
[KZ1]  2ØØ: ' Bildschirm
[U82]  2Ø1: aus.monitor:
[ØE6]  2Ø2: IF quelldat$ <> "" THEN GOTO monitor.con
[EKE]  2Ø3: LOCATE 8, 4: PRINT "Sie haben noch keine SA
            M-Datei angegeben. Drücken Sie bitte eine T
            aste !"
[5H3]  2Ø4: WHILE INKEY$ = "": WEND
[VT4]  2Ø5: LOCATE 8, 1: PRINT STRING$(8Ø, 32)
[AV6]  2Ø6: GOTO eingabe:'Zurück ins Hauptprogramm
[E2Ø]  2Ø7: '
[S72]  2Ø8: monitor.con:
[PNØ]  2Ø9: CLS
[WT4]  21Ø: OPEN quelldat$ FOR INPUT AS #1
[MF1]  211: zeilen = Ø
[3E2]  212: WHILE NOT EOF(1)
[VB3]  213:   LINE INPUT #1, PZEILE$
[8Ø3]  214:   zeilen = zeilen + 1
[TF5]  215:   PRINT zeilen; : PRINT ": "; PZEILE$
[TRØ]  216: WEND
[3WØ]  217: CLOSE
[6YD]  218: PRINT : PRINT "Die Ausgabe der "; zeilen; :
             PRINT " Zeile"; : IF zeilen <> 1 THEN PRIN
            T "n";
[TFL]  219: PRINT " der SAM-Datei auf dem Bildschirm is
            t beendet.": PRINT "                    Drücken
             Sie bitte eine Taste, um fortzufahren.";
[5H3]  22Ø: WHILE INKEY$ = "": WEND
[WS1]  221: GOTO menue
[E2Ø]  222: '
[UT6]  223: 'Ausgabe des Quelltextes auf den Drucker
[R72]  224: aus.drucker:
[9B6]  225: IF quelldat$ <> "" THEN GOTO drucker.con
[EKE]  226: LOCATE 8, 4: PRINT "Sie haben noch keine SA
            M-Datei angegeben. Drücken Sie bitte eine T
            aste !"
[5H3]  227: WHILE INKEY$ = "": WEND
[VT4]  228: LOCATE 8, 1: PRINT STRING$(8Ø, 32)
[AV6]  229: GOTO eingabe:'Zurück ins Hauptprogramm
[E2Ø]  23Ø: '
[N62]  231: drucker.con:
[PNØ]  232: CLS
[WT4]  233: OPEN quelldat$ FOR INPUT AS #1
[P86]  234: zeilen = Ø: Seite = 1: AZEILEN = Ø: flag =
            Ø
[E2Ø]  235: '
[WN5]  236: WHILE NOT EOF(1):'Schließe Dateien
[VB3]  237:   LINE INPUT #1, PZEILE$
[PH1]  238: druck.w1:
[W75]  239:   IF AZEILEN <> Ø THEN GOTO druck.w2
[X66]  24Ø:   A1 = INT((br.max - 8Ø) / 2): A2 = br.max
            - 8Ø - A1
[UNU]  241:   LPRINT : LPRINT "Listing einer SAM-Datei
                        "; STRING$(A1, 32); "Datum: ";
             MID$(DATE$, 4, 2); "."; LEFT$(DATE$, 2); "
            ."; RIGHT$(DATE$, 4); "                    "; S
            TRING$(A2, 32); "Seite: "; Seite
[7F6]  242:   LPRINT : LPRINT "SAM-Datei:"; quelldat$
[HD7]  243:   LPRINT STRING$(br.max, "-"): LPRINT : AZE
            ILEN = 6
[RH1]  244: druck.w2:
[PL4]  245:   IF flag = Ø THEN GOTO druck.w3
[PW7]  246:   LPRINT "        "; LEFT$(PZEILE$, 73): zei
            len = zeilen + 1
[152]  247:   GOTO druck.w4
[SH1]  248: druck.w3:
[8Ø3]  249:   zeilen = zeilen + 1
[N97]  25Ø:   LPRINT zeilen; : LPRINT ": "; LEFT$(PZEIL
            E$, 73)
[TH1]  251: druck.w4:
[V1B]  252:   IF LEN(PZEILE$) > 73 THEN flag = -1: PZEI
            LE$ = MID$(PZEILE$, 74) ELSE flag = Ø
[873]  253:   AZEILEN = AZEILEN + 1
[Ø7D]  254:   IF AZEILEN = ln.max - 3 THEN LPRINT : LPR
            INT : LPRINT : AZEILEN = Ø: Seite = Seite +
             1
[P54]  255:   IF flag THEN GOTO druck.w1
[TRØ]  256: WEND
[E2Ø]  257: '
[3WØ]  258: CLOSE
[4L3]  259: FOR q = 1 TO 72 - zeilen
[H51]  26Ø:   LPRINT
[K21]  261: NEXT q
[3AF]  262: PRINT : PRINT : PRINT "   Die Ausgabe der "
            ; zeilen; : PRINT " Zeile"; : IF zeilen <>
            1 THEN PRINT "n";
[WVL]  263: PRINT " der SAM-Datei auf dem Drucker ist b
            eendet.": PRINT : PRINT "                D
            rücken Sie bitte eine Taste, um fortzufahre
            n.";
[5H3]  264: WHILE INKEY$ = "": WEND
[WS1]  265: GOTO menue
[E2Ø]  266: '
[U84]  267: 'Erzeugen einer EXE-Datei
[LT1]  268: assembler:
[Ø45]  269: IF DOS THEN GOTO Hauptprogramm
[UY5]  27Ø: IF quelldat$ <> "" THEN GOTO assem.con
[EKE]  271: LOCATE 8, 4: PRINT "Sie haben noch keine SA
            M-Datei angegeben. Drücken Sie bitte eine T
            aste !"
[5H3]  272: WHILE INKEY$ = "": WEND
[VT4]  273: LOCATE 8, 1: PRINT STRING$(8Ø, 32)
[AV6]  274: GOTO eingabe:'Zurück ins Hauptprogramm
[E2Ø]  275: '
[NP1]  276: assem.con:
[RG6]  277: IF zieldat$ <> "" THEN GOTO Hauptprogramm
[E2Ø]  278: '
[A93]  279: zieldat$ = quelldat$
[VP2]  28Ø: P = LEN(zieldat$)
[9E4]  281: WHILE MID$(zieldat$, P, 1) <> "."
[A21]  282:   P = P - 1
[TRØ]  283: WEND
[UE5]  284: zieldat$ = LEFT$(zieldat$, P) + "exe"
[P93]  285: GOTO Hauptprogramm
[E2Ø]  286: '
[8Y6]  287: 'Zweiter Lauf (verarbeitet INCLUDE-Dateien
[L67]  288: ' und fügt eventuell vorhandene MAKROS ein)
[ØU1]  289: lauf.zwei:
[ZCA]  29Ø: OPEN "work5.wrk" FOR INPUT AS #1: OPEN "wor
            k2.wrk" FOR OUTPUT AS #4
[W41]  291: lauf = Ø
[343]  292: aktsymbol = vordef
[XU1]  293: codezeig = Ø
[WV1]  294: zeilnum = Ø
[3E2]  295: WHILE NOT EOF(1)
[2A6]  296:   GOSUB lies.zeile:'Einlesen einer Zeile
[KM5]  297:   GOSUB analyse:'Zeile übersetzen
[E2Ø]  298:   '
[B16]  299:   'Falls INCLUDE-Anweisungen vorhanden
[1M6]  3ØØ:   ' sind, werden sie, so weit als möglich,
[442]  3Ø1:   ' ausgeführt
[6H6]  3Ø2:   IF befehl$ <> "INCLUDE" THEN GOTO do.befe
            hl
[277]  3Ø3:   datei$ = MID$(zeile$, INSTR(zeile$, befeh
            l$) + 8)
[SN3]  3Ø4:   WHILE INSTR(datei$, ";")
[4B6]  3Ø5:     datei$ = LEFT$(datei$, INSTR(datei$, ";
            ") - 1)
[TRØ]  3Ø6:   WEND
[5T3]  3Ø7:   WHILE LEFT$(datei$, 1) = " "
[3D3]  3Ø8:     datei$ = MID$(datei$, 2)
[TRØ]  3Ø9:   WEND
[X14]  31Ø:   WHILE RIGHT$(datei$, 1) = " "
[YC5]  311:     datei$ = LEFT$(datei$, LEN(datei$) - 1)
[TRØ]  312:   WEND
[R94]  313:   OPEN datei$ FOR INPUT AS #5
[E25]  314:   IF EOF(5) THEN GOTO fehl.include
[E2Ø]  315:   '
[LE2]  316:   WHILE NOT EOF(5)
[LE2]  317:     LINE INPUT #5, a$
[GN1]  318:     PRINT #4, a$
[TRØ]  319:   WEND
[451]  32Ø:   CLOSE #5
[ME1]  321:   zeile$ = ""
[EE2]  322:   GOTO m.schluss
[E2Ø]  323:   '
[U46]  324:   'Falls DO-Anweisungen vorhanden sind,
[CL5]  325:   ' werden sie so weit als möglich
[442]  326:   ' ausgeführt
[DL1]  327: do.befehl:
[3N5]  328:   IF befehl$ <> "DO" THEN GOTO m.schluss
[TZ7]  329:   parameter$ = MID$(zeile$, INSTR(zeile$, b
            efehl$) + 3)
[ME4]  33Ø:   WHILE INSTR(parameter$, ";")
[ML8]  331:     parameter$ = LEFT$(parameter$, INSTR(pa
            rameter$, ";") - 1)
[TRØ]  332:   WEND
[1K4]  333:   WHILE LEFT$(parameter$, 1) = " "
[LZ4]  334:     parameter$ = MID$(parameter$, 2)
[TRØ]  335:   WEND
[BU4]  336:   WHILE RIGHT$(parameter$, 1) = " "
[6M7]  337:     parameter$ = LEFT$(parameter$, LEN(para
            meter$) - 1)
[TRØ]  338:   WEND
[E43]  339:   FOR zaehler = 1 TO 8
[HV2]  34Ø:     param$(zaehler) = ""
[G42]  341:   NEXT zaehler
[E2Ø]  342:   '
[PD1]  343:   MNAME$ = ""
[6HD]  344:   IF INSTR(parameter$, ",") = Ø THEN MNAME$
             = parameter$: parameter$ = "": GOTO m.para
            meter
[9S7]  345:   MNAME$ = LEFT$(parameter$, INSTR(paramete
            r$, ",") - 1)
[DM8]  346:   parameter$ = MID$(parameter$, INSTR(param
            eter$, ",") + 1)
[E2Ø]  347:   '
[Y32]  348: m.parameter:
```

Listing 1. Die dritte SAM-Version »sam3.asc« läßt sich mit SBasic, Quick Basic und Turbo Basic compilieren

```
[5HH]  349:   IF MNAME$ = "" THEN PRINT : PRINT "Beim A
              ufruf des Makros fehlt die Angabe des Makro
              namens.": GOTO macro.fehler
[A4B]  35Ø:   IF parameter$ = "" THEN zaehler = Ø: GOTO
               parameter.ok ELSE zaehler = 1
[E2Ø]  351:   '
[892]  352: m.auswertung:
[RPC]  353:   IF INSTR(parameter$, ",") = Ø THEN param$
              (zaehler) = parameter$: GOTO parameter.ok
[HG9]  354:   param$(zaehler) = LEFT$(parameter$, INSTR
              (parameter$, ",") - 1)
[DM8]  355:   parameter$ = MID$(parameter$, INSTR(param
              eter$, ",") + 1)
[RF9]  356:   IF zaehler < 8 THEN zaehler = zaehler + 1
              : GOTO m.auswertung
[GFS]  357:   PRINT : PRINT "Die Anzahl der übergebenen
               Parameter ist zu hoch oder die Form der Pa
              rameter    falsch beim Aufruf des Makros: "
              ; MNAME$: GOTO do.fehler
[E2Ø]  358:   '
[YW4]  359:   'Form des Makroaufrufs stimmt
[L82]  36Ø: parameter.ok:
[685]  361:   IF marke$ <> "" THEN PRINT #4, marke$
[M95]  362:   OPEN "makros.wrk" FOR INPUT AS #6
[DX8]  363:   IF EOF(6) THEN GOTO do.fehler:'Fehler bei
               einem Do-Aufruf
[E2Ø]  364:   '
[VA2]  365: oeffne.mdatei:
[P73]  366:   LINE INPUT #6, DUMMY$
[8F2]  367:   INPUT #6, werte
[GB3]  368:   LINE INPUT #6, SPNAME$
[TX5]  369:   IF SPNAME$ = MNAME$ THEN GOTO makro.da
[BG2]  37Ø: suche.schluss:
[863]  371:   LINE INPUT #6, zeile$
[SN9]  372:   IF zeile$ <> "Makrodefinition: Ende" THEN
               GOTO suche.schluss
[ZR5]  373:   IF NOT EOF(6) THEN GOTO oeffne.mdatei
[551]  374:   CLOSE #6
[VX2]  375:   fehler = fehler + 1
[ME1]  376:   zeile$ = ""
[ØTF]  377:   PRINT #2, "==> Fehler: Das aufgerufene Ma
              kro "; MNAME$; " wurde noch nicht definiert
              .": GOTO m.schluss
[8E1]  378: makro.da:
[BF6]  379:   IF zaehler <> werte THEN warnung = warnun
              g + 1: PRINT #2, "==> Warnung: Anzahl der P
              arameter bei Definition und Aufruf stimmen
              nicht       überein. Überzählige werden ign
              oriert, Zeilen mit nicht angegebenen Parame
              tern  gelöscht. Makro: ";
[E2Ø]  38Ø:   '
[5Z1]  381: lies.makro:
[863]  382:   LINE INPUT #6, zeile$
[NKH]  383:   IF zeile$ = "Makrodefinition: Ende" THEN
              CLOSE #6: zeile$ = "": GOTO m.schluss:'Ende
               und Zeile nicht abspeichern
[E2Ø]  384:   '
[6Y7]  385:   FOR ZAEHLER2 = 1 TO 8:'Höchstens acht Par
              ameter
[F76]  386:     'Wenn Parameter nicht vorhanden ist,
[9W5]  387:     ' dann weitermachen; Parameter durch
[6F6]  388:     ' angegebenen Wert ersetzen; falls kein
[3T6]  389:     ' Parameter angegeben, dann Zeile lösch
              en
[UE8]  39Ø:     WHILE INSTR(zeile$, "%" + RIGHT$(STR$(Z
              AEHLER2), 1)) <> Ø
[RLP]  391:       zeile$ = LEFT$(zeile$, INSTR(zeile$,
              "%" + RIGHT$(STR$(ZAEHLER2), 1)) - 1) + par
              am$(ZAEHLER2) + MID$(zeile$, INSTR(zeile$,
              "%" + RIGHT$(STR$(ZAEHLER2), 1)) + 2)
[MW5]  392:       IF param$(ZAEHLER2) = "" THEN zeile$
              = ""
[TRØ]  393:     WEND
[M72]  394:   NEXT ZAEHLER2
[285]  395:   IF zeile$ <> "" THEN PRINT #4, zeile$
[MS2]  396:   GOTO lies.makro
[2T1]  397: m.schluss:
[285]  398:   IF zeile$ <> "" THEN PRINT #4, zeile$
[5N2]  399:   GOSUB aus.param
[TRØ]  4ØØ: WEND
[351]  4Ø1: CLOSE #4
[651]  4Ø2: RETURN
[E2Ø]  4Ø3: '
[LY5]  4Ø4: 'Hauptprogramm, das die benötigten
[C74]  4Ø5: ' Unterprogramme aufruft
[3K2]  4Ø6: Hauptprogramm:
[PNØ]  4Ø7: CLS
[6BA]  4Ø8: GOSUB flex.init:     'Initialisierung der be
              nötigten Variablen
[4FA]  4Ø9: GOSUB lauf.eins:     'Lauf 1: Vorbereitung d
              er Makro-Definitionen
[8L8]  41Ø: GOSUB lauf.zwei:     'Lauf 2: 'INCLUDE'und V
              oranpassung
[8B8]  411: GOSUB symbol:        'Lauf 3: Aufbau der Sym
              boltabelle
[X19]  412: IF KEINENDE THEN CLOSE : IF DOS THEN END EL
              SE GOTO menue
[G8C]  413: GOSUB gen.exe:       'Lauf 4: Erzeugen der E
              XE-Datei und der Assemblerlistings
[DV7]  414: GOSUB assem.schluss:'Beenden der Übersetzun
              g
[WS1]  415: GOTO menue
[E2Ø]  416: '
[5H6]  417: 'Dritter Lauf (Aufbau der Symboltabelle)
[HC1]  418: symbol:
[VB6]  419: CLOSE #1: OPEN "work2.wrk" FOR INPUT AS #1
[GG2]  42Ø: KEINENDE = wahr
[X41]  421: lauf = 1
[XU1]  422: codezeig = Ø
[WV1]  423: zeilnum = Ø
[3E2]  424: WHILE NOT EOF(1)
[2A6]  425:   GOSUB lies.zeile:'Einlesen einer Zeile
[6U5]  426:   GOSUB analyse:    'Zeile übersetzen
[EA6]  427:   IF befehl$ = "ENDS" THEN GOTO schluss.seg
[E2Ø]  428:   '
[VØ6]  429:   'Eintrag des Symbols in die Tabelle
[T56]  43Ø:   IF marke$ <> "" THEN GOSUB symboleintrag
[D16]  431:   IF befehl$ <> "" THEN GOSUB trans.befehl
[1HB]  432:   IF marke$ <> "" OR befehl$ <> "" THEN IF
              NOT KEINENDE THEN GOTO kein.schluss
[2Ø6]  433:   IF befehl$ = "END" THEN KEINENDE = falsch
[362]  434: schluss.seg:
[5N2]  435:   GOSUB aus.param
[TRØ]  436: WEND
[E2Ø]  437: '
[L56]  438: IF NOT KEINENDE THEN GOTO sym.schluss
[A92]  439: kein.schluss:
[69Y]  44Ø: PRINT #2, : PRINT #2, " Der END-Befehl wurd
              e zu früh oder gar nicht benutzt. SAM III u
              nterbricht die": PRINT #2, "       Übersetz
              ung der SAM-Datei. Drücken Sie bitte eine b
              eliebige Taste !"
[5H3]  441: WHILE INKEY$ = "": WEND
[GG2]  442: KEINENDE = wahr
[R82]  443: sym.schluss:
[651]  444: RETURN
[E2Ø]  445: '
[BV6]  446: 'Vierter Lauf (Erzeugen der EXE-Datei und
[G24]  447: ' des Assemblerlistings)
[3D1]  448: gen.exe:
[LM2]  449: GOSUB init.drei
[E2Ø]  45Ø: '
[3E2]  451: WHILE NOT EOF(1)
[2A6]  452:   GOSUB lies.zeile:'Einlesen einer Zeile
[KM5]  453:   GOSUB analyse:'Zeile übersetzen
[ØBB]  454:   IF marke$ <> "" THEN GOSUB Phasenfehler:'
              Überprüfen auf Phasenfehler
[D16]  455:   IF befehl$ <> "" THEN GOSUB trans.befehl
[T4A]  456:   GOSUB code.ausgabe:'Ausgabe des Assembler
              codes und -listings
[5N2]  457:   GOSUB aus.param
[M97]  458:   taste$ = INKEY$: IF taste$ = "" THEN GOTO
               weiter
[PR3]  459:   WHILE INKEY$ <> "": WEND
[5H3]  46Ø:   WHILE INKEY$ = "": WEND
[XA1]  461: weiter:
[TRØ]  462: WEND
[E2Ø]  463: '
[651]  464: RETURN
[E2Ø]  465: '
[LV6]  466: 'Unterprogramm zum Einlesen einer Zeile
[GZ1]  467: lies.zeile:
[Y53]  468: LINE INPUT #1, zeile$
[MC3]  469: zeilnum = zeilnum + 1
[1E3]  47Ø: offsetbedarf = nichts
[AF2]  471: dsflag = falsch
[HG2]  472: befehllaenge = Ø
[E2Ø]  473: '
[651]  474: RETURN
[E2Ø]  475: '
[L86]  476: 'Unterprogramm, in dem die eingelesene
[1X3]  477: ' Zeile übersetzt wird
[ND1]  478: analyse:
[TØ2]  479: zeilzeig1 = 1
[VØ2]  48Ø: zeilzeig2 = 1
[HE1]  481: marke$ = ""
[SK1]  482: befehl$ = ""
[GF1]  483: quell$ = ""
[NA1]  484: ziel$ = ""
[MN4]  485: endzeig = INSTR(zeile$, ";") - 1
[E2Ø]  486: '
[P96]  487: IF endzeig = -1 THEN endzeig = LEN(zeile$)
[KT5]  488: IF endzeig = Ø THEN GOTO ana.schluss
[SD9]  489: GOSUB klein.gross:'Klein- in Großbuchstaben
               umwandeln
[E2Ø]  49Ø: '
[AR8]  491: IF INSTR(trenn$, LEFT$(zeile$, 1)) THEN GOT
              O op.analyse
[4Ø2]  492: GOSUB Aufbau
[9Ø2]  493: marke$ = FELD$
[UV9]  494: IF RIGHT$(marke$, 1) = ":" THEN marke$ = LE
              FT$(marke$, LEN(marke$) - 1)
[E2Ø]  495: '
[UT3]  496: 'Analyse des Operanden
[CY1]  497: op.analyse:
[4Ø2]  498: GOSUB Aufbau
[L36]  499: IF NOT gefunden THEN GOTO ana.schluss
```

```
[Z72]  5ØØ: befehl$ = FELD$
[AB3]  5Ø1: opndzeig = zeilzeig1
[4Ø2]  5Ø2: GOSUB Aufbau
[L36]  5Ø3: IF NOT gefunden THEN GOTO ana.schluss
[2Y1]  5Ø4: ziel$ = FELD$
[4Ø2]  5Ø5: GOSUB Aufbau
[L36]  5Ø6: IF NOT gefunden THEN GOTO ana.schluss
[812]  5Ø7: quell$ = FELD$
[252]  5Ø8: ana.schluss:
[E2Ø]  5Ø9: '
[651]  51Ø: RETURN
[E2Ø]  511: '
[3C7]  512: 'Unterprogramm, das die in der eingelesenen
[3P6]  513: ' Zeile vorhandenen Kleinbuchstaben in
[1W4]  514: ' Großbuchstaben umwandelt
[842]  515: klein.gross:
[163]  516: FOR i = 1 TO endzeig
[P63]  517:   b$ = MID$(zeile$, i, 1)
[9VF]  518:   IF b$ = "'" THEN ZEICHENDE = INSTR(i + 1,
       zeile$, b$): IF ZEICHENDE > Ø THEN i = ZEI
       CHENDE: GOTO kommentar
[AAB]  519:   IF ASC(b$) >= 97 AND ASC(b$) <= 122 THEN
       b$ = CHR$(ASC(b$) - 32): MID$(zeile$, i, 1)
       = b$
[VU1]  52Ø: kommentar:
[U11]  521: NEXT i
[E2Ø]  522: '
[651]  523: RETURN
[E2Ø]  524: '
[YD6]  525: 'Unterprogramm, das den Befehlsaufbau
[UØ2]  526: ' analysiert
[4A1]  527: Aufbau:
[BB4]  528: WHILE zeilzeig1 <= endzeig
[H4A]  529:   IF INSTR(trenn$, MID$(zeile$, zeilzeig1,
       1)) = Ø THEN GOTO marke.1
[8W3]  53Ø:   zeilzeig1 = zeilzeig1 + 1
[TRØ]  531: WEND
[E2Ø]  532: '
[281]  533: marke.1:
[BM6]  534: IF zeilzeig1 <= endzeig THEN GOTO marke.2
[PP2]  535: gefunden = falsch
[651]  536: RETURN
[E2Ø]  537: '
[681]  538: marke.2:
[4S7]  539: IF MID$(zeile$, zeilzeig1, 1) <> "'" THEN G
       OTO marke.3
[ØU6]  54Ø: ZEICHENDE = INSTR(zeilzeig1 + 1, zeile$, "'
       ")
[F55]  541: IF ZEICHENDE = Ø THEN GOTO marke.3
[PW3]  542: zeilzeig2 = ZEICHENDE + 1
[4A3]  543: GOTO feldzuweisung
[E2Ø]  544: '
[A81]  545: marke.3:
[WF3]  546: zeilzeig2 = zeilzeig1
[FB4]  547: WHILE zeilzeig2 <= endzeig
[1EB]  548:   IF INSTR(trenn$, MID$(zeile$, zeilzeig2,
       1)) > Ø THEN GOTO feldzuweisung
[CW3]  549:   zeilzeig2 = zeilzeig2 + 1
[TRØ]  55Ø: WEND
[E2Ø]  551: '
[CM2]  552: feldzuweisung:
[8F8]  553: FELD$ = MID$(zeile$, zeilzeig1, zeilzeig2 -
       zeilzeig1)
[WF3]  554: zeilzeig1 = zeilzeig2
[4G2]  555: gefunden = wahr
[651]  556: RETURN
[E2Ø]  557: '
[7Ø7]  558: 'Unterprogramm, das ein bisher noch nicht
[MN7]  559: ' definiertes Symbol in die Tabelle einträg
       t
[3L2]  56Ø: symboleintrag:
[ZG2]  561: objekt$ = marke$
[A73]  562: GOSUB suche.symbol
[7N6]  563: IF NOT gefunden THEN GOTO sym.ueberlauf
[VX2]  564: fehler = fehler + 1
[2GD]  565: PRINT #2, "==> Fehler: Label "; marke$; " i
       n "; zeilnum; "zum wiederholten Mal definie
       rt."
[651]  566: RETURN
[E2Ø]  567: '
[MF2]  568: sym.ueberlauf:
[P88]  569: IF aktsymbol < symbolzahl THEN GOTO sym.ein
       tragen
[VX2]  57Ø: fehler = fehler + 1
[58C]  571: PRINT #2, "==> Fehler: Zu viele benutzerdef
       inierte Symbole in Zeile"; zeilnum
[651]  572: RETURN
[E2Ø]  573: '
[PG2]  574: sym.eintragen:
[YØ4]  575: aktsymbol = aktsymbol + 1
[TC4]  576: symbol$(aktsymbol) = marke$
[HL4]  577: wert1(aktsymbol) = codezeig
[MF4]  578: symboltyp(aktsymbol) = nah
[651]  579: RETURN
[E2Ø]  58Ø: '
[Y27]  581: 'Unterprogramm, das die Symboltabelle auf
[E64]  582: ' Phasenfehler untersucht
[E72]  583: Phasenfehler:
```

```
[LP9]  584: IF befehl$ = "EQU" OR befehl$ = "ENDS" THEN
       GOTO phasen.schluss
[ZG2]  585: objekt$ = marke$
[A73]  586: GOSUB suche.symbol
[K9D]  587: IF (symboltyp(tabzeiger) AND (nah OR speich
       er)) = falsch THEN GOTO phasen.schluss
[VB9]  588: IF wert1(tabzeiger) = codezeig THEN GOTO ph
       asen.schluss
[E2Ø]  589: '
[HP8]  59Ø: fehler = fehler + 1: PRINT #2, "==> Fehler:
       Phasenfehler"
[SN2]  591: phasen.schluss:
[E2Ø]  592: '
[651]  593: RETURN
[E2Ø]  594: '
[347]  595: 'Unterprogramm, das die angegebene Adresse
[UH4]  596: ' in der Symboltabelle sucht
[E82]  597: suche.symbol:
[TØ5]  598: FOR tabzeiger = 1 TO aktsymbol
[Z3B]  599:   IF symbol$(tabzeiger) = objekt$ THEN GOTO
       symbol.da:'Symbol gefunden
[FG2]  6ØØ: NEXT tabzeiger
[PP2]  6Ø1: gefunden = falsch
[651]  6Ø2: RETURN
[E2Ø]  6Ø3: '
[NP1]  6Ø4: symbol.da:
[4G2]  6Ø5: gefunden = wahr
[651]  6Ø6: RETURN
[E2Ø]  6Ø7: '
[S56]  6Ø8: 'Unterprogramm, das den angegebenen
[VN6]  6Ø9: ' Assemblerbefehl in der Kommandotabelle
[X71]  61Ø: ' sucht
[F82]  611: befehlssuche:
[ZS2]  612: MOVE = befehlzahl
[RD2]  613: START = MOVE / 2
[E2Ø]  614: '
[Y92]  615: WHILE MOVE >= 2
[2A2]  616:   MOVE = MOVE / 2
[KLC]  617:   IF befehl$ > kommando$(START) THEN START
       = START + MOVE ELSE START = START - MOVE
[654]  618:   IF START < 1 THEN START = 1
[X37]  619:   IF START > befehlzahl THEN START = befehl
       zahl
[TRØ]  62Ø: WEND
[E2Ø]  621: '
[DV5]  622: FOR operzeig = START TO befehlzahl
[AA8]  623:   IF kommando$(operzeig) > befehl$ THEN GOT
       O da.falsch
[Ø69]  624:   IF kommando$(operzeig) <> befehl$ THEN GO
       TO such.schluss
[3ZB]  625:   IF (quellregtyp(operzeig) AND quelltyp) =
       falsch THEN GOTO such.schluss
[XLB]  626:   IF (zielregtyp(operzeig) AND zieltyp) = f
       alsch THEN GOTO such.schluss
[V12]  627:   GOTO da.wahr
[CA2]  628: such.schluss:
[EB2]  629: NEXT operzeig
[E2Ø]  63Ø: '
[WL1]  631: da.falsch:
[PP2]  632: gefunden = falsch
[651]  633: RETURN
[E2Ø]  634: '
[P91]  635: da.wahr:
[4G2]  636: gefunden = wahr
[651]  637: RETURN
[E2Ø]  638: '
[8V5]  639: 'Unterprogramm, das die angegebene
[GS3]  64Ø: ' Operation übersetzt
[852]  641: trans.befehl:
[S5N]  642: IF befehl$ = "TITLE" OR befehl$ = "SUBTTL"
       OR befehl$ = "EVEN" OR befehl$ = "ENDS" OR
       befehl$ = "SEGMENT" OR befehl$ = "END" THEN
       GOTO bef.suche
[E2Ø]  643: '
[WC2]  644: objekt$ = ziel$
[BV2]  645: GOSUB op.auswert
[YW2]  646: zieltyp = objtyp
[2E3]  647: zielwert1 = objwert1
[5E3]  648: zielwert2 = objwert2
[E2Ø]  649: '
[78C]  65Ø: IF befehl$ = "RET" THEN quelltyp = prozedur
       typ(stapelzeiger): GOTO bef.suche
[ZN5]  651: objekt$ = quell$: GOSUB op.auswert
[UX2]  652: quelltyp = objtyp
[PF3]  653: quellwert1 = objwert1
[VF3]  654: quellwert2 = objwert2
[E2Ø]  655: '
[V36]  656: 'Suche Kommando in der Befehlstabelle
[AP1]  657: bef.suche:
[9K2]  658: objekt$ = befehl$
[953]  659: GOSUB befehlssuche
[U26]  66Ø: IF gefunden THEN GOTO flag.zuweisung
[E2Ø]  661: '
[ZR3]  662: IF lauf = 1 THEN RETURN
[HEA]  663: fehler = fehler + 1: PRINT #2, "==> Fehler:
       Syntax-Fehler: "; befehl$
[E2Ø]  664: '
```

Listing 1. Die dritte SAM-Version »sam3.asc« läßt sich mit SBasic, Quick Basic und Turbo Basic compilieren

```
[35G]  665: IF ((akku8 OR akku16 OR reg8 OR reg16 OR se
           greg OR cs) AND (zieltyp OR quelltyp)) THEN
            GOTO s.schluss
[1SC]  666: IF (quelltyp AND (nichts OR direkt8 OR dire
           kt16)) = falsch THEN GOTO s.schluss
[DU8]  667: IF INSTR("BW", RIGHT$(befehl$, 1)) <> Ø THE
           N GOTO s.schluss
[E2Ø]  668: '
[UC3]  669: warnung = warnung + 1
[5FC]  67Ø: PRINT #2, "==> Warnung: Achten Sie auf Wort
           - beziehungsweise Byte-Operationen"
[ET1]  671: s.schluss:
[E2Ø]  672: '
[651]  673: RETURN
[E2Ø]  674: '
[FT2]  675: flag.zuweisung:
[BA3]  676: flag = flag(operzeig)
[1ØA]  677: IF flag AND operand THEN GOSUB gen.code ELS
           E GOSUB pseudo.ops
[651]  678: RETURN
[E2Ø]  679: '
[FR5]  68Ø: 'Unterprogramm, das den Operanden
[ZY1]  681: ' auswertet
[L22]  682: op.auswert:
[WØ6]  683: IF LEN(objekt$) > Ø THEN GOTO obj.suche
[8N2]  684: objtyp = nichts
[651]  685: RETURN
[E2Ø]  686: '
[BR1]  687: obj.suche:
[A73]  688: GOSUB suche.symbol
[KN5]  689: IF NOT gefunden THEN GOTO w.objekt
[8Y4]  69Ø: objtyp = symboltyp(tabzeiger)
[RK4]  691: objwert1 = wert1(tabzeiger)
[739]  692: IF tabzeiger <= vordef THEN objwert2 = wert
           2(tabzeiger)
[651]  693: RETURN
[E2Ø]  694: '
[DG1]  695: w.objekt:
[U62]  696: GOSUB op.zahl
[E75]  697: IF NOT gefunden THEN GOTO w2.obj
[1T2]  698: objtyp = zahltyp
[6B3]  699: objwert1 = zahlwert
[651]  7ØØ: RETURN
[E2Ø]  7Ø1: '
[N41]  7Ø2: w2.obj:
[C83]  7Ø3: GOSUB speichertest
[F75]  7Ø4: IF NOT gefunden THEN GOTO w3.obj
[1V2]  7Ø5: objtyp = speicher
[AR3]  7Ø6: objwert1 = SPEICHERADR
[651]  7Ø7: RETURN
[E2Ø]  7Ø8: '
[S41]  7Ø9: w3.obj:
[SR2]  71Ø: GOSUB op.offset
[G75]  711: IF NOT gefunden THEN GOTO w4.obj
[KD3]  712: objtyp = speicherreg
[D23]  713: objwert1 = regwert
[651]  714: RETURN
[E2Ø]  715: '
[V41]  716: w4.obj:
[6W2]  717: GOSUB opisoffset
[H75]  718: IF NOT gefunden THEN GOTO w5.obj
[8W2]  719: objtyp = OFFSETYP
[MK3]  72Ø: objwert1 = OFFSETWERT
[651]  721: RETURN
[E2Ø]  722: '
[Y41]  723: w5.obj:
[ZS2]  724: GOSUB zeichen.op
[K75]  725: IF NOT gefunden THEN GOTO w6.obj
[5P4]  726: objtyp = direkt8 OR direkt16
[7C3]  727: objwert1 = zeichwert
[651]  728: RETURN
[E2Ø]  729: '
[151]  73Ø: w6.obj:
[587]  731: IF LEFT$(objekt$, 1) <> "'" THEN GOTO sym.f
           ehler
[AN2]  732: objtyp = zeichen
[651]  733: RETURN
[E2Ø]  734: '
[222]  735: sym.fehler:
[ØRE]  736: IF lauf = 2 THEN PRINT #2, "==> Fehler: Sym
           bol "; objekt$; " nicht definiert": fehler
           = fehler + 1
[1Ø4]  737: objtyp = nah OR speicher
[E2Ø]  738: '
[651]  739: RETURN
[E2Ø]  74Ø: '
[BL6]  741: 'Unterprogramm, das überprüft, ob der
[VS3]  742: ' Operand eine Zahl ist
[4B1]  743: op.zahl:
[PP2]  744: gefunden = falsch
[5Y1]  745: tn$ = objekt$
[WV5]  746: IF RIGHT$(tn$, 1) <> "H" THEN GOTO binaer
[E2Ø]  747: '
[G54]  748: tn$ = LEFT$(tn$, LEN(tn$) - 1)
[8LØ]  749: i = 1
[E2Ø]  75Ø: '
[863]  751: FOR i = 1 TO LEN(tn$)
[ME2]  752:   b$ = MID$(tn$, i, 1)
```

```
[CX6]  753:   IF INSTR("Ø123456789ABCDEF", b$) = Ø THEN
           RETURN
[U11]  754: NEXT i
[E2Ø]  755: '
[MP3]  756: zahlwert = VAL("&H" + tn$)
[X22]  757: GOTO zahl.ok
[E2Ø]  758: '
[ØA1]  759: binaer:
[1L6]  76Ø: IF RIGHT$(tn$, 1) <> "B" THEN GOTO zahl.tra
           ns
[G54]  761: tn$ = LEFT$(tn$, LEN(tn$) - 1)
[8LØ]  762: i = 1
[E2Ø]  763: '
[863]  764: FOR i = 1 TO LEN(tn$)
[ME2]  765:   b$ = MID$(tn$, i, 1)
[PS4]  766:   IF INSTR("Ø1", b$) = Ø THEN RETURN
[U11]  767: NEXT i
[E2Ø]  768: '
[3X1]  769: zahlwert = Ø
[E2Ø]  77Ø: '
[WB4]  771: FOR i = LEN(tn$) TO 1 STEP -1
[UT8]  772:   zahlwert = zahlwert + VAL(MID$(tn$, i, 1)
           ) * 2 ^ (LEN(tn$) - i)
[U11]  773: NEXT i
[E2Ø]  774: '
[X22]  775: GOTO zahl.ok
[122]  776: zahl.trans:
[E2Ø]  777: '
[863]  778: FOR i = 1 TO LEN(tn$)
[ME2]  779:   b$ = MID$(tn$, i, 1)
[L46]  78Ø:   IF INSTR("Ø123456789-+", b$) = Ø THEN RET
           URN
[U11]  781: NEXT i
[E2Ø]  782: '
[FY2]  783: zahlwert = VAL(tn$)
[KC1]  784: zahl.ok:
[4G2]  785: gefunden = wahr
[M4D]  786: IF LEN(HEX$(zahlwert)) < 3 THEN zahltyp = d
           irekt16 OR direkt8 ELSE zahltyp = direkt16
[E2Ø]  787: '
[651]  788: RETURN
[E2Ø]  789: '
[2M5]  79Ø: 'Unterprogramm, das den direkten
[BU4]  791: ' Speicherzugriff auswertet
[2A2]  792: speichertest:
[3Y1]  793: MR$ = objekt$
[SP8]  794: IF LEFT$(MR$, 1) <> "[" OR RIGHT$(MR$, 1) <
           > "]" THEN RETURN
[E2Ø]  795: '
[ZN4]  796: objekt$ = MID$(MR$, 2, LEN(MR$) - 2)
[U62]  797: GOSUB op.zahl
[5H9]  798: IF gefunden THEN SPEICHERADR = zahlwert: GO
           TO ok.speicher
[A73]  799: GOSUB suche.symbol
[1U6]  8ØØ: IF NOT gefunden THEN GOTO falsch.speicher
[73C]  8Ø1: IF (symboltyp(tabzeiger) AND direkt16) = fa
           lsch THEN GOTO falsch.speicher
[615]  8Ø2: SPEICHERADR = wert1(tabzeiger)
[SU2]  8Ø3: GOTO ok.speicher
[E2Ø]  8Ø4: '
[KR2]  8Ø5: falsch.speicher:
[PP2]  8Ø6: gefunden = falsch
[VX1]  8Ø7: objekt$ = MR$
[651]  8Ø8: RETURN
[E2Ø]  8Ø9: '
[122]  81Ø: ok.speicher:
[VX1]  811: objekt$ = MR$
[651]  812: RETURN
[E2Ø]  813: '
[5A6]  814: 'Unterprogramm, das den Operanden als
[216]  815: ' Offset zu einem Register behandelt
[UP1]  816: op.offset:
[ZM2]  817: pdoder$ = objekt$
[KPD]  818: IF objekt$ = "[BP]" THEN regwert = 6: offse
           tbedarf = direkt8: offset = Ø: GOTO ein.sym
           bol
[984]  819: zeiger = INSTR(objekt$, "[")
[6T5]  82Ø: IF zeiger <= 1 THEN GOTO null.symbol
[KN7]  821: regis$ = RIGHT$(pdoder$, LEN(pdoder$) - zei
           ger + 1)
[GK7]  822: IF regis$ = "[BP]" THEN regwert = 6: GOTO o
           p.disp
[LK2]  823: objekt$ = regis$
[A73]  824: GOSUB suche.symbol
[E2Ø]  825: '
[MRC]  826: IF NOT gefunden OR symboltyp(tabzeiger) <>
           speicherreg THEN GOTO null.symbol
[FC4]  827: regwert = wert1(tabzeiger)
[FC1]  828: op.disp:
[3Y4]  829: disp$ = LEFT$(pdoder$, zeiger - 1)
[CB2]  83Ø: objekt$ = disp$
[A73]  831: GOSUB suche.symbol
[E2Ø]  832: '
[526]  833: IF NOT gefunden THEN GOTO kein.symbol
[D8D]  834: IF (symboltyp(tabzeiger) AND (direkt16 OR d
           irekt8)) = falsch THEN GOTO kein.symbol
[D16]  835: offsetbedarf = symboltyp(tabzeiger)
[CØ4]  836: offset = wert1(tabzeiger)
```

```
[2U2]  837: GOTO ein.symbol
[E2Ø]  838: '
[S52]  839: kein.symbol:
[U62]  84Ø: GOSUB op.zahl
[UZC]  841: IF gefunden THEN offsetbedarf = zahltyp: of
            fset = zahlwert: GOTO ein.symbol
[6W2]  842: GOSUB opisoffset
[136]  843: IF NOT gefunden THEN GOTO null.symbol
[F14]  844: offsetbedarf = OFFSETYP
[NC3]  845: offset = OFFSETWERT
[E2Ø]  846: '
[W22]  847: ein.symbol:
[9M2]  848: objekt$ = pdoder$
[4G2]  849: gefunden = wahr
[E2Ø]  85Ø: '
[651]  851: RETURN
[E2Ø]  852: '
[S72]  853: null.symbol:
[9M2]  854: objekt$ = pdoder$
[PP2]  855: gefunden = falsch
[651]  856: RETURN
[E2Ø]  857: '
[5A6]  858: 'Unterprogramm, das den Operanden als
[4A3]  859: ' Offset deklariert
[E22]  86Ø: opisoffset:
[AY1]  861: os$ = objekt$
[CE9]  862: IF LEFT$(os$, 7) <> "OFFSET(" THEN gefunden
             = falsch: RETURN
[194]  863: IF lauf = 1 THEN GOTO arg.ok
[H36]  864: objekt$ = MID$(objekt$, 8, LEN(objekt$) - 8
            )
[A73]  865: GOSUB suche.symbol
[E2Ø]  866: '
[AHH]  867: IF gefunden AND (symboltyp(tabzeiger) AND (
            speicher OR nah)) THEN OFFSETWERT = wert1(t
            abzeiger): GOTO arg.ok
[VX2]  868: fehler = fehler + 1
[S9B]  869: PRINT #2, "==> Fehler: Unerlaubtes oder und
            efiniertes Offset-Argument"
[F92]  87Ø: OFFSETWERT = Ø
[E2Ø]  871: '
[991]  872: arg.ok:
[4G2]  873: gefunden = wahr
[A63]  874: OFFSETYP = direkt16
[4Y1]  875: objekt$ = os$
[651]  876: RETURN
[E2Ø]  877: '
[5A6]  878: 'Unterprogramm, das den Operanden als
[493]  879: ' Zeichen definiert
[512]  88Ø: zeichen.op:
[PP2]  881: gefunden = falsch
[SV4]  882: IF LEN(objekt$) <> 3 THEN RETURN
[SY5]  883: IF LEFT$(objekt$, 1) <> "'" THEN RETURN
[UU5]  884: IF RIGHT$(objekt$, 1) <> "'" THEN RETURN
[E2Ø]  885: '
[4G2]  886: gefunden = wahr
[LG5]  887: zeichwert = ASC(MID$(objekt$, 2, 1))
[651]  888: RETURN
[E2Ø]  889: '
[ZN6]  89Ø: 'Unterprogramm, das im zweiten Lauf den
[TP3]  891: ' Programmcode erzeugt
[ZF1]  892: gen.code:
[7S2]  893: GOSUB wort.byte
[3L3]  894: codezeig = codezeig + 1
[CB5]  895: IF lauf = 2 THEN GOSUB code.aufbau
[G5C]  896: IF (operwert(operzeig) <> 213) AND (operwer
            t(operzeig) <> 212) THEN GOTO gen.m1
[E2Ø]  897: '
[3L3]  898: codezeig = codezeig + 1
[UØC]  899: IF lauf = 2 THEN befehllaenge = befehllaeng
            e + 1: befehltab(befehllaenge) = 1Ø
[E2Ø]  9ØØ: '
[S61]  9Ø1: gen.m1:
[NH6]  9Ø2: IF offsetbedarf = nichts THEN GOTO gen.m2
[GMD]  9Ø3: IF (offsetbedarf AND direkt8) THEN codezeig
             = codezeig + 1 ELSE codezeig = codezeig +
            2
[E2Ø]  9Ø4: '
[V61]  9Ø5: gen.m2:
[64B]  9Ø6: IF (flag AND (modbytebedarf OR erweibedarf)
            ) = falsch THEN GOTO gen.m3
[ERA]  9Ø7: IF (zieltyp OR quelltyp) AND speicher THEN
            codezeig = codezeig + 2
[E2Ø]  9Ø8: '
[Y61]  9Ø9: gen.m3:
[458]  91Ø: IF (flag AND erweibedarf) = falsch THEN GOT
            O gen.m4
[3L3]  911: codezeig = codezeig + 1
[BY4]  912: IF lauf = 2 THEN GOSUB erw.byte
[E2Ø]  913: '
[171]  914: gen.m4:
[EA8]  915: IF (flag AND modbytebedarf) = falsch THEN G
            OTO gen.m5
[3L3]  916: codezeig = codezeig + 1
[TL4]  917: IF lauf = 2 THEN GOSUB adr.art
[E2Ø]  918: '
[471]  919: gen.m5:
[438]  92Ø: IF (flag AND disp8bedarf) = falsch THEN GOT
            O gen.m6
```

```
[3L3]  921: codezeig = codezeig + 1
[EP5]  922: IF lauf = 2 THEN GOSUB diff.relativ
[E2Ø]  923: '
[771]  924: gen.m6:
[EØ8]  925: IF (flag AND disp16bedarf) = falsch THEN GO
            TO gen.m7
[7L3]  926: codezeig = codezeig + 2
[A74]  927: IF lauf = 2 THEN GOSUB dis16
[E2Ø]  928: '
[A71]  929: gen.m7:
[Y78]  93Ø: IF (flag AND direkt8bedarf) = falsch THEN G
            OTO gen.m8
[3L3]  931: codezeig = codezeig + 1
[PN4]  932: IF lauf = 2 THEN GOSUB byte.di
[E2Ø]  933: '
[D71]  934: gen.m8:
[ZY9]  935: IF wort OR ((flag AND direkt16bedarf) = fal
            sch) THEN GOTO gen.m9
[3L3]  936: codezeig = codezeig + 1
[PN4]  937: IF lauf = 2 THEN GOSUB byte.di
[E2Ø]  938: '
[G71]  939: gen.m9:
[T2B]  94Ø: IF NOT (wort) OR ((flag AND direkt16bedarf)
             = falsch) THEN GOTO gen.m1Ø
[W8C]  941: IF zieltyp = direkt16 THEN codezeig = codez
            eig + 4 ELSE codezeig = codezeig + 2
[DR4]  942: IF lauf = 2 THEN GOSUB wort.di
[E2Ø]  943: '
[A71]  944: gen.m1Ø:
[RT9]  945: IF (flag AND speicherbedarf) = falsch THEN
            GOTO gen.schluss
[7L3]  946: codezeig = codezeig + 2
[Z25]  947: IF lauf = 2 THEN GOSUB baut.16bit
[E2Ø]  948: '
[762]  949: gen.schluss:
[651]  95Ø: RETURN
[E2Ø]  951: '
[HW6]  952: 'Unterprogramme, das zwischen Wort- und
[K35]  953: ' Byte-Operanden unterscheidet
[9U1]  954: wort.byte:
[UYC]  955: IF (zieltyp OR quelltyp) AND (reg16 OR akku
            16 OR segreg OR cs) THEN GOTO wort.ja
[1DA]  956: IF (zieltyp OR quelltyp) AND (reg8 OR akku8
            ) THEN GOTO wort.nein
[D27]  957: IF RIGHT$(befehl$, 1) = "B" THEN GOTO wort.
            nein
[E2Ø]  958: '
[FD1]  959: wort.ja:
[7X1]  96Ø: wort = wahr
[651]  961: RETURN
[E2Ø]  962: '
[4T1]  963: wort.nein:
[T32]  964: wort = falsch
[651]  965: RETURN
[E2Ø]  966: '
[VC6]  967: 'Unterprogramm, das die Pseudobefehle
[8W1]  968: ' behandelt
[ZZ1]  969: pseudo.ops:
[V5L]  97Ø: ON operwert(operzeig) GOSUB ps.equ, ps.org,
             ps.db, ps.ds, ps.proc, ps.endp, ps.title,
            ps.even, ps.end, ps.segment, ps.ends
[XLD]  971: '                                  EQU     ORG
             DB      DS      PROC      ENDP      TITLE
            EVEN     END     SEGMENT     ENDS
[651]  972: RETURN
[E2Ø]  973: '
[225]  974: 'Pseudobefehl 'END' (erfüllt keinen
[673]  975: ' besonderen Zweck)
[671]  976: ps.end:
[651]  977: RETURN
[E2Ø]  978: '
[E93]  979: 'Pseudobefehl 'EVEN'
[3C1]  98Ø: ps.even:
[YM9]  981: IF codezeig / 2 <> INT(codezeig / 2) THEN c
            odezeig = codezeig + 1
[651]  982: RETURN
[E2Ø]  983: '
[7B6]  984: 'Unterprogramm, das den Programmcode
[UA1]  985: ' aufbaut
[P22]  986: code.aufbau:
[ZX4]  987: befehllaenge = befehllaenge + 1
[MB7]  988: befehltab(befehllaenge) = operwert(operzeig
            )
[E2Ø]  989: '
[9S7]  99Ø: IF (flag AND regsum) = falsch THEN GOTO auf
            bau.m2
[8U9]  991: IF zieltyp AND (segreg OR cs) THEN r = ziel
            wert2: GOTO aufbau.m1
[GKA]  992: IF (flag AND richtung) THEN r = quellwert2
            / 8 ELSE r = zielwert2 / 8
[E2Ø]  993: '
[HL1]  994: aufbau.m1:
[SD8]  995: befehltab(befehllaenge) = befehltab(befehll
            aenge) + r
[E2Ø]  996: '
[LL1]  997: aufbau.m2:
```

Listing 1. Die dritte SAM-Version »sam3.asc« läßt sich mit SBasic, Quick Basic und Turbo Basic compilieren

```
[PN7]  998: IF (flag AND konst2) = falsch THEN GOTO auf
           bau.m3
[8DA]  999: IF wort THEN befehltab(befehllaenge) = befe
           hltab(befehllaenge) + 1
[E2Ø] 1ØØØ: '
[NL1] 1ØØ1: aufbau.m3:
[MP8] 1ØØ2: IF (flag AND konst1) = falsch THEN GOTO auf
           bau.schluss
[6AC] 1ØØ3: IF quelltyp AND c1 THEN befehltab(befehllae
           nge) = befehltab(befehllaenge) + 2
[E2Ø] 1ØØ4: '
[HP2] 1ØØ5: aufbau.schluss:
[651] 1ØØ6: RETURN
[E2Ø] 1ØØ7: '
[277] 1ØØ8: 'Unterprogramm, das das Erweiterungsbyte
[B61] 1ØØ9: ' bildet
[2N1] 1Ø1Ø: erw.byte:
[YF1] 1Ø11: maske = 56
[T33] 1Ø12: ext = flag AND maske
[RFA] 1Ø13: IF flag AND richtung THEN zielwert2 = ext E
           LSE quellwert2 = ext
[M32] 1Ø14: GOSUB adr.art
[E2Ø] 1Ø15: '
[651] 1Ø16: RETURN
[E2Ø] 1Ø17: '
[G47] 1Ø18: 'Unterprogramm, das die Adressierungsart
[8W1] 1Ø19: ' behandelt
[6E1] 1Ø2Ø: adr.art:
[ZX4] 1Ø21: befehllaenge = befehllaenge + 1
[Z8B] 1Ø22: IF ((zieltyp OR quelltyp) AND speicher) = f
           alsch THEN GOTO adr.weiter
[ME9] 1Ø23: IF zieltyp = speicher THEN m = quellwert2 E
           LSE m = zielwert2
[ZU4] 1Ø24: befehltab(befehllaenge) = 6 + m
[2M2] 1Ø25: GOSUB baut.16bit
[E2Ø] 1Ø26: '
[651] 1Ø27: RETURN
[E2Ø] 1Ø28: '
[112] 1Ø29: adr.weiter:
[9VD] 1Ø3Ø: IF flag AND richtung THEN m = quellwert1 +
           zielwert2 ELSE m = zielwert1 + quellwert2
[HD4] 1Ø31: befehltab(befehllaenge) = m
[E2Ø] 1Ø32: '
[XK7] 1Ø33: IF offsetbedarf = nichts THEN GOTO adr.schl
           uss
[W67] 1Ø34: IF offset > 127 OR offset < -128 THEN GOTO
           adr.w2
[WG8] 1Ø35: befehltab(befehllaenge) = befehltab(befehll
           aenge) + 64
[596] 1Ø36: IF offset < Ø THEN offset = offset AND 255
[ZX4] 1Ø37: befehllaenge = befehllaenge + 1
[C75] 1Ø38: befehltab(befehllaenge) = offset
[E2Ø] 1Ø39: '
[651] 1Ø4Ø: RETURN
[E2Ø] 1Ø41: '
[G71] 1Ø42: adr.w2:
[CR8] 1Ø43: befehltab(befehllaenge) = befehltab(befehll
           aenge) + 128
[3Y4] 1Ø44: befehllaenge = befehllaenge + 2
[CW4] 1Ø45: lsb = offset: GOSUB teil.16bit
[4Ø5] 1Ø46: befehltab(befehllaenge - 1) = lsb
[8M4] 1Ø47: befehltab(befehllaenge) = msb
[862] 1Ø48: adr.schluss:
[E2Ø] 1Ø49: '
[651] 1Ø5Ø: RETURN
[E2Ø] 1Ø51: '
[AB7] 1Ø52: 'Unterprogramm, das eine 16-Bit-Zahl in den
[TK7] 1Ø53: ' niedrigwertigen und höherwertigen Bereich
[C22] 1Ø54: ' unterteilt
[6V1] 1Ø55: teil.16bit:
[E2Ø] 1Ø56: '
[ZV1] 1Ø57: h$ = HEX$(lsb)
[KE4] 1Ø58: h$ = STRING$(4 - LEN(h$), "Ø") + h$
[Z54] 1Ø59: lsb = VAL("&H" + RIGHT$(h$, 2))
[KU3] 1Ø6Ø: msb = VAL("&H" + LEFT$(h$, 2))
[E2Ø] 1Ø61: '
[651] 1Ø62: RETURN
[E2Ø] 1Ø63: '
[PY6] 1Ø64: 'Unterprogramm, das die Differenz für die
[Ø35] 1Ø65: ' relative Adressierung bildet
[Z62] 1Ø66: diff.relativ:
[GU3] 1Ø67: d = zielwert1 - codezeig
[XU4] 1Ø68: IF ABS(d) < 128 THEN GOTO dist.ok
[WKØ] 1Ø69: d = Ø
[E2Ø] 1Ø7Ø: '
[FAG] 1Ø71: IF lauf = 2 THEN PRINT #2, "==> Fehler: Dis
           tanz zu groß für einen 'short jump'": fehle
           r = fehler + 1
[KD1] 1Ø72: dist.ok:
[E2Ø] 1Ø73: '
[PS3] 1Ø74: IF d < Ø THEN d = d AND 255
[ZX4] 1Ø75: befehllaenge = befehllaenge + 1
[DC4] 1Ø76: befehltab(befehllaenge) = d
[E2Ø] 1Ø77: '
[651] 1Ø78: RETURN
[E2Ø] 1Ø79: '
[WW5] 1Ø8Ø: 'Routine, die das 16-Bit-Displacement
[PW1] 1Ø81: ' errechnet
[UZØ] 1Ø82: dis16:
[GU3] 1Ø83: d = zielwert1 - codezeig
```

```
[S2H] 1Ø84: IF befehl$ = "JMP" AND d <= 128 THEN PRINT
           #2, "==> Warnung: Sie könnten 'JMPS' benutz
           en": warnung = warnung + 1
[Ø37] 1Ø85: IF d >= Ø OR befehl$ = "CALL" THEN GOTO rue
           ck16
[WKØ] 1Ø86: d = Ø
[E2Ø] 1Ø87: '
[4BH] 1Ø88: IF lauf = 2 THEN PRINT #2, "==> Fehler: Nic
           ht erlaubter Rückwärtssprung ('long jump')"
           : fehler = fehler + 1
[N91] 1Ø89: rueck16:
[E2Ø] 1Ø9Ø: '
[5W3] 1Ø91: lsb = d: GOSUB teil.16bit
[3Y4] 1Ø92: befehllaenge = befehllaenge + 2
[4Ø5] 1Ø93: befehltab(befehllaenge - 1) = lsb
[8M4] 1Ø94: befehltab(befehllaenge) = msb
[E2Ø] 1Ø95: '
[651] 1Ø96: RETURN
[E2Ø] 1Ø97: '
[2M5] 1Ø98: 'Unterprogramm, das den direkten
[2T3] 1Ø99: ' Wortzugriff beachtet
[XD1] 11ØØ: wort.di:
[WPB] 11Ø1: IF zieltyp AND direkt16 THEN direktwert = z
           ielwert1: GOSUB wort.weiter
[B6C] 11Ø2: IF quelltyp AND direkt16 THEN direktwert =
           quellwert1: GOSUB wort.weiter
[E2Ø] 11Ø3: '
[651] 11Ø4: RETURN
[E2Ø] 11Ø5: '
[E72] 11Ø6: wort.weiter:
[2M5] 11Ø7: lsb = direktwert: GOSUB teil.16bit
[3Y4] 11Ø8: befehllaenge = befehllaenge + 2
[4Ø5] 11Ø9: befehltab(befehllaenge - 1) = lsb
[8M4] 111Ø: befehltab(befehllaenge) = msb
[E2Ø] 1111: '
[651] 1112: RETURN
[E2Ø] 1113: '
[2M5] 1114: 'Unterprogramm, das den direkten
[FN3] 1115: ' Bytezugriff beachtet
[AD1] 1116: byte.di:
[PBB] 1117: IF zieltyp AND direkt8 THEN direktwert = zi
           elwert1: GOSUB byte.weiter
[REC] 1118: IF quelltyp AND direkt8 THEN direktwert = q
           uellwert1: GOSUB byte.weiter
[E2Ø] 1119: '
[651] 112Ø: RETURN
[E2Ø] 1121: '
[V62] 1122: byte.weiter:
[KE9] 1123: IF direktwert <= 255 AND direktwert >= Ø TH
           EN GOTO daten.ok
[GA2] 1124: direktwert = Ø
[E2Ø] 1125: '
[6ZC] 1126: IF lauf = 2 THEN fehler = fehler + 1: PRINT
            #2, "==> Fehler: Datenanweisung zu lang"
[1G1] 1127: daten.ok:
[E2Ø] 1128: '
[ZX4] 1129: befehllaenge = befehllaenge + 1
[2Z5] 113Ø: befehltab(befehllaenge) = direktwert
[651] 1131: RETURN
[E2Ø] 1132: '
[FL4] 1133: 'Bildet eine 16-Bit-Adresse
[EV1] 1134: baut.16bit:
[6DA] 1135: IF zieltyp = speicher THEN lsb = zielwert1
           ELSE lsb = quellwert1
[YM2] 1136: GOSUB teil.16bit
[3Y4] 1137: befehllaenge = befehllaenge + 2
[4Ø5] 1138: befehltab(befehllaenge - 1) = lsb
[8M4] 1139: befehltab(befehllaenge) = msb
[E2Ø] 114Ø: '
[651] 1141: RETURN
[E2Ø] 1142: '
[H43] 1143: 'Pseudobefehl 'EQU'
[Z81] 1144: ps.equ:
[FZ4] 1145: IF (marke$ <> "") THEN GOTO equ.w1
[74H] 1146: IF lauf = 2 THEN fehler = fehler + 1: PRINT
            #2, "==> Fehler: Die Symbolzuweisung fehlt
            bei der EQU-Anweisung"
[651] 1147: RETURN
[E2Ø] 1148: '
[891] 1149: equ.w1:
[T85] 115Ø: IF lauf = 2 THEN GOTO equ.schluss
[XD7] 1151: IF zieltyp <> (nah OR speicher) THEN GOTO e
           qu.w2
[VX2] 1152: fehler = fehler + 1
[ZVE] 1153: PRINT #2, "==> Fehler: EQU betrifft in Zeil
           e"; zeilnum; "eine noch nicht definierte Zu
           weisung"
[651] 1154: RETURN
[E2Ø] 1155: '
[B91] 1156: equ.w2:
[SS4] 1157: wert1(aktsymbol) = zielwert1
[D95] 1158: symboltyp(aktsymbol) = zieltyp
[382] 1159: equ.schluss:
[E2Ø] 116Ø: '
[651] 1161: RETURN
[E2Ø] 1162: '
[L33] 1163: 'Pseudobefehl 'ORG'
[181] 1164: ps.org:
[GB3] 1165: codezeig = zielwert1
```

```
[651] 1166: RETURN
[E2Ø] 1167: '
[CU2] 1168: 'Pseudobefehl 'DB'
[7YØ] 1169: ps.db:
[754] 117Ø: IF lauf = 2 THEN GOTO db.w1
[Y88] 1171: IF marke$ <> "" THEN symboltyp(aktsymbol) =
            speicher
[E2Ø] 1172: '
[DVØ] 1173: db.w1:
[447] 1174: zeilzeig1 = opndzeig: zeilzeig2 = opndzeig
[P54] 1175: WHILE zeilzeig1 < endzeig
[4Ø2] 1176:   GOSUB Aufbau
[PZ5] 1177:   IF NOT gefunden THEN GOTO db.schluss
[2N4] 1178:   objekt$ = FELD$: GOSUB op.zahl
[5EA] 1179:   IF NOT gefunden OR (zahltyp AND direkt8)
            = falsch THEN GOTO db.w2
[3W1] 118Ø:   GOSUB db.w4
[BU2] 1181:   GOTO op.bekannt
[E2Ø] 1182:   '
[FVØ] 1183: db.w2:
[UM5] 1184:   IF LEFT$(FELD$, 1) <> "'" THEN GOTO db.w3
[7W1] 1185:   GOSUB db.w5
[BU2] 1186:   GOTO op.bekannt
[E2Ø] 1187:   '
[HVØ] 1188: db.w3:
[8GD] 1189:   IF lauf = 2 THEN PRINT #2, "==> Fehler: U
            nbekannter Operand "; FELD$: fehler = fehle
            r + 1
[5Y1] 119Ø: op.bekannt:
[TRØ] 1191: WEND
[E2Ø] 1192: '
[3Y1] 1193: db.schluss:
[DE5] 1194: codezeig = codezeig + befehllaenge
[651] 1195: RETURN
[E2Ø] 1196: '
[LVØ] 1197: db.w4:
[ZX4] 1198: befehllaenge = befehllaenge + 1
[VP5] 1199: befehltab(befehllaenge) = zahlwert
[651] 12ØØ: RETURN
[E2Ø] 12Ø1: '
[NVØ] 12Ø2: db.w5:
[F45] 12Ø3: FELD$ = MID$(FELD$, 2, LEN(FELD$) - 2)
[E2Ø] 12Ø4: '
[8F3] 12Ø5: FOR i = 1 TO LEN(FELD$)
[ZX4] 12Ø6:   befehllaenge = befehllaenge + 1
[L37] 12Ø7:   befehltab(befehllaenge) = ASC(MID$(FELD$,
            i, 1))
[U11] 12Ø8: NEXT i
[E2Ø] 12Ø9: '
[651] 121Ø: RETURN
[E2Ø] 1211: '
[EV2] 1212: 'Pseudobefehl 'DS'
[9ZØ] 1213: ps.ds:
[E12] 1214: dsflag = wahr
[S54] 1215: IF lauf = 2 THEN GOTO ds.w1
[Y88] 1216: IF marke$ <> "" THEN symboltyp(aktsymbol) =
            speicher
[E2Ø] 1217: '
[ØXØ] 1218: ds.w1:
[3KD] 1219: IF quelltyp AND direkt8 THEN quellzielwert
            = quellwert1 ELSE quellzielwert = Ø
[S54] 122Ø: IF lauf = 1 THEN GOTO ds.w2
[E2Ø] 1221: '
[XH3] 1222: FOR i = 1 TO zielwert1
[P19] 1223:   LSET byte$ = CHR$(quellzielwert): d.z = d
            .z + 1: PUT #3, d.z
[U11] 1224: NEXT i
[E2Ø] 1225: '
[2XØ] 1226: ds.w2:
[365] 1227: codezeig = codezeig + zielwert1
[651] 1228: RETURN
[E2Ø] 1229: '
[393] 123Ø: 'Pseudobefehl 'PROC'
[SB1] 1231: ps.proc:
[288] 1232: IF stapelzeiger < stapelgroesse THEN GOTO p
            roc.w2
[PC4] 1233: IF lauf = 1 THEN GOTO proc.w1
[E2Ø] 1234: '
[VX2] 1235: fehler = fehler + 1
[DH9] 1236: PRINT #2, "==> Fehler: Prozeduren sind zu t
            ief verschachtelt"
[MB1] 1237: proc.w1:
[E2Ø] 1238: '
[651] 1239: RETURN
[E2Ø] 124Ø: '
[SB1] 1241: proc.w2:
[U65] 1242: stapelzeiger = stapelzeiger + 1
[9E6] 1243: prozedurtyp(stapelzeiger) = zieltyp
[651] 1244: RETURN
[E2Ø] 1245: '
[393] 1246: 'Pseudobefehl 'ENDP'
[SB1] 1247: ps.endp:
[XV5] 1248: IF stapelzeiger > Ø THEN GOTO endp.w2
[RB4] 1249: IF lauf = 1 THEN GOTO endp.w1
[E2Ø] 125Ø: '
[VX2] 1251: fehler = fehler + 1
[1N5] 1252: PRINT #2, "==> Fehler: ENDP ohne PROC"
[E2Ø] 1253: '
[L91] 1254: endp.w1:
[651] 1255: RETURN
[E2Ø] 1256: '
[R91] 1257: endp.w2:
[Y65] 1258: stapelzeiger = stapelzeiger - 1
[651] 1259: RETURN
[E2Ø] 126Ø: '
[EV3] 1261: 'Pseudobefehl 'SEGMENT'
[WZ1] 1262: ps.segment:
[PØ6] 1263: IF lauf = 1 THEN GOTO segment.schluss
[E2Ø] 1264: '
[NL2] 1265: segname$ = marke$
[TWM] 1266: IF segmente = 1 THEN fehler = fehler + 1: P
            RINT "==> Fehler: Unzulässiger Segmentanfan
            g": GOTO segment.schluss ELSE segmente = 1
[A97] 1267: pruef$ = MID$(zeile$, INSTR(zeile$, befehl$
            ) + 7)
[E2Ø] 1268: '
[TZ1] 1269: segment.w1:
[4D6] 127Ø: IF pruef$ = "" THEN GOTO segment.schluss
[Y3C] 1271: IF INSTR(trenn$, LEFT$(pruef$, 1)) THEN pru
            ef$ = MID$(pruef$, 2): GOTO segment.w1
[YSE] 1272: IF INSTR(trenn$, RIGHT$(pruef$, 1)) THEN pr
            uef$ = LEFT$(pruef$, LEN(pruef$) - 1): GOTO
             segment.w1
[SF7] 1273: IF pruef$ = "CODE" THEN code = codezeig: c2
             = wahr
[B88] 1274: IF pruef$ = "STACK" THEN stapel = codezeig:
             s2 = wahr
[MY2] 1275: segment.schluss:
[E2Ø] 1276: '
[651] 1277: RETURN
[E2Ø] 1278: '
[4Z5] 1279: 'Pseudobefehle 'TITEL' und 'SUBTITLE'
[WG1] 128Ø: ps.title:
[E2Ø] 1281: '
[Ø4B] 1282: IF befehl$ = "TITLE" THEN title$ = MID$(zei
            le$, INSTR(zeile$, befehl$) + 6)
[ØDB] 1283: IF befehl$ = "SUBTTL" THEN subttl$ = MID$(z
            eile$, INSTR(zeile$, befehl$) + 7)
[E2Ø] 1284: '
[651] 1285: RETURN
[E2Ø] 1286: '
[F93] 1287: 'Pseudobefehl 'ENDS'
[4C1] 1288: ps.ends:
[E85] 1289: IF lauf = 1 THEN GOTO ens.schluss
[YN5] 129Ø: IF marke$ = segname$ THEN GOTO ens.w1
[E2Ø] 1291: '
[9PH] 1292: fehler = fehler + 1: PRINT #2, "==> Fehler:
             Das Segmentende hat nicht den gleichen Nam
            en wie der Segmentanfang"
[E2Ø] 1293: '
[R81] 1294: ens.w1:
[S2G] 1295: IF segmente = Ø THEN fehler = fehler + 1: P
            RINT #2, "==> Fehler: Falsches Segmentende"
             ELSE segmente = Ø
[K72] 1296: ens.schluss:
[E2Ø] 1297: '
[651] 1298: RETURN
[E2Ø] 1299: '
[217] 13ØØ: 'Unterprogramm, das den Programmcode und
[Ø15] 13Ø1: ' das Assemblerlisting ausgibt
[Ø52] 13Ø2: code.ausgabe:
[E2Ø] 13Ø3: '
[M5D] 13Ø4: IF dsflag THEN h$ = HEX$(codezeig - zielwer
            t1) ELSE h$ = HEX$(codezeig - befehllaenge)
[KE4] 13Ø5: h$ = STRING$(4 - LEN(h$), "Ø") + h$
[8V5] 13Ø6: IF asslis <> 4 THEN PRINT #2, TAB(1); h$;
[8LØ] 13Ø7: i = 1
[2B5] 13Ø8: IF asslis <> 4 THEN PRINT #2, TAB(6);
[E2Ø] 13Ø9: '
[9S1] 131Ø: WHILE i <= 6
[BA6] 1311:   IF i > befehllaenge THEN GOTO ausgabe.w1
[5S4] 1312:   LSET byte$ = CHR$(befehltab(i))
[LE3] 1313:   d.z = d.z + 1: PUT #3, d.z
[3V2] 1314:   laenge = laenge + 1
[XA3] 1315:   h$ = HEX$(befehltab(i))
[WV3] 1316:   IF LEN(h$) = 1 THEN h$ = "Ø" + h$
[SS4] 1317:   IF asslis <> 4 THEN PRINT #2, h$;
[611] 1318:   i = i + 1
[TRØ] 1319: WEND
[E2Ø] 132Ø: '
[7Y1] 1321: ausgabe.w1:
[BF5] 1322: IF asslis <> 4 THEN PRINT #2, TAB(19);
[TU5] 1323: IF asslis <> 4 THEN PRINT #2, zeilnum;
[E2Ø] 1324: '
[CB9] 1325: IF asslis = 3 THEN PRINT #2, SPACE$(2); zei
            le$: GOTO aus.byte
[VM9] 1326: IF asslis <> 4 THEN PRINT #2, SPACE$(2); LE
            FT$(zeile$, LWIDTH - 26)
[E2Ø] 1327: '
[LM1] 1328: aus.byte:
[E2Ø] 1329: '
[RS3] 133Ø: WHILE i <= befehllaenge
[L18] 1331:   IF i MOD 6 = 1 THEN IF asslis <> 4 THEN P
            RINT #2, TAB(6);
[5S4] 1332:   LSET byte$ = CHR$(befehltab(i))
[LE3] 1333:   d.z = d.z + 1: PUT #3, d.z
[3V2] 1334:   laenge = laenge + 1
```

Listing 1. Die dritte SAM-Version »sam3.asc« läßt sich mit SBasic, Quick Basic und Turbo Basic compilieren

```
[XA3] 1335:    h$ = HEX$(befehltab(i))
[WV3] 1336:    IF LEN(h$) = 1 THEN h$ = "Ø" + h$
[SS4] 1337:    IF asslis <> 4 THEN PRINT #2, h$;
[611] 1338:    i = i + 1
[TRØ] 1339: WEND
[E2Ø] 134Ø: '
[X98] 1341: IF befehllaenge > 6 THEN IF asslis <> 4 THE
            N PRINT #2,
[651] 1342: RETURN
[E2Ø] 1343: '
[BX5] 1344: 'Initialisierung des dritten Laufs
[9R1] 1345: init.drei:
[E2Ø] 1346: '
[Ø51] 1347: CLOSE #1
[VØ5] 1348: OPEN "work2.wrk" FOR INPUT AS #1
[Y41] 1349: lauf = 2
[XU1] 135Ø: codezeig = Ø
[WV1] 1351: zeilnum = Ø
[E2Ø] 1352: '
[651] 1353: RETURN
[E2Ø] 1354: '
[8V5] 1355: 'Beenden der Übersetzungsroutine
[EH2] 1356: assem.schluss:
[E2Ø] 1357: '
[8PD] 1358: IF stapelzeiger > Ø THEN fehler = fehler +
            1: PRINT #2, "==> Fehler: Es fehlt ein 'END
            p' "
[UEF] 1359: IF c2 = falsch THEN warnung = warnung + 1:
            PRINT #2, "==> Warnung: Kein Code-Segment s
            pezifiziert"
[6XF] 136Ø: IF s2 = falsch THEN warnung = warnung + 1:
            PRINT #2, "==> Warnung: Kein Stapel-Segment
             spezifiziert"
[T9E] 1361: IF segmente = 1 THEN fehler = fehler + 1: P
            RINT #2, "==> Fehler: Kein Segmentende ange
            geben"
[E2Ø] 1362: '
[691] 1363: PRINT #2,
[691] 1364: PRINT #2,
[C86] 1365: PRINT #2, fehler; "Fehler festgestellt."
[237] 1366: PRINT #2, warnung; "Warnung(en) ausgegeben.
            "
[E2Ø] 1367: '
[862] 1368: GOSUB sym.aus
[E2Ø] 1369: '
[B1A] 137Ø: PRINT #2, : PRINT #2, laenge; "Bytes für di
            e EXE-Datei erzeugt."
[8N6] 1371: IF l$ = "lpt1:" THEN PRINT #2, CHR$(27); CH
            R$(64)
[3WØ] 1372: CLOSE
[E2Ø] 1373: '
[S26] 1374: IF fehler = Ø THEN GOSUB exe.struktur
[KF2] 1375: SHELL "del *.wrk"
[RNØ] 1376: BEEP
[GXØ] 1377: PRINT
[GXØ] 1378: PRINT
[ND6] 1379: PRINT "SAM III hat seine Arbeit beendet."
[GXØ] 138Ø: PRINT
[E2Ø] 1381: '
[4MØ] 1382: END
[E2Ø] 1383: '
[PD4] 1384: 'Ausgabe der Symboltabelle
[UG1] 1385: sym.aus:
[MX2] 1386: PRINT #2, : PRINT #2,
[8Z3] 1387: PRINT #2, "Symboltabelle:"
[HK3] 1388: PRINT #2, "==============="
[E2Ø] 1389: '
[5Ø2] 139Ø: i = vordef + 1
[MC3] 1391: perzeil = LWIDTH \ 19
[BRE] 1392: IF i > aktsymbol THEN PRINT #2, : PRINT #2,
             "Es wurden keine Symbole definiert.": GOTO
             sym.w1
[E2Ø] 1393: '
[L93] 1394: WHILE i <= aktsymbol
[VP2] 1395:    h$ = HEX$(wert1(i))
[KE4] 1396:    h$ = STRING$(4 - LEN(h$), "Ø") + h$
[BT4] 1397:    PRINT #2, symbol$(i); " "; h$; "    ";
[611] 1398:    i = i + 1
[SØ7] 1399:    IF (i - vordef) MOD perzeil = 1 THEN PRIN
            T #2,
[TRØ] 14ØØ: WEND
[E2Ø] 14Ø1: '
[W91] 14Ø2: sym.w1:
[691] 14Ø3: PRINT #2,
[651] 14Ø4: RETURN
[E2Ø] 14Ø5: '
[KE5] 14Ø6: 'Angabe der aktuellen Parameter
[FT1] 14Ø7: aus.param:
[E2Ø] 14Ø8: '
[XD3] 14Ø9: X = POS(Ø): Y = CSRLIN
[HA3] 141Ø: LOCATE 25, 1: COLOR Ø, 7
[2WE] 1411: PRINT "Durchlauf"; lauf + 2; "ist aktiv.
                                Anzahl der bearbeiteten Ze
            ilen: "; zeilnum;
[GG3] 1412: COLOR 7, Ø: LOCATE Y, X
[E2Ø] 1413: '
[651] 1414: RETURN
[E2Ø] 1415: '
[375] 1416: 'Unterprogramm für EXE-Dateien
```

```
[ZD2] 1417: exe.struktur:
[E29] 1418: OPEN "work.wrk" FOR RANDOM AS #3 LEN = 1: F
            IELD #3, 1 AS byte$
[XY8] 1419: OPEN zieldat$ FOR RANDOM AS #2 LEN = 1: FIE
            LD #2, 1 AS ZBYTE$
[M11] 142Ø: d.z = Ø
[E2Ø] 1421: '
[6X6] 1422: LSET ZBYTE$ = CHR$(77): d.z = d.z + 1: PUT
            #2, d.z
[UW6] 1423: LSET ZBYTE$ = CHR$(9Ø): d.z = d.z + 1: PUT
            #2, d.z
[D3A] 1424: LSET ZBYTE$ = CHR$((laenge MOD 512) MOD 256
            ): d.z = d.z + 1: PUT #2, d.z
[PCA] 1425: LSET ZBYTE$ = CHR$(INT((laenge MOD 512) / 2
            56)): d.z = d.z + 1: PUT #2, d.z
[AYC] 1426: LSET ZBYTE$ = CHR$(INT(laenge / 512) + 1 +
            SGN(laenge MOD 512) * 1): d.z = d.z + 1: PU
            T #2, d.z
[HH6] 1427: LSET ZBYTE$ = CHR$(Ø): d.z = d.z + 1: PUT #
            2, d.z
[L1A] 1428: LSET ZBYTE$ = CHR$(Ø): d.z = d.z + 1: PUT #
            2, d.z: d.z = d.z + 1: PUT #2, d.z
[SW6] 1429: LSET ZBYTE$ = CHR$(32): d.z = d.z + 1: PUT
            #2, d.z
[NHD] 143Ø: LSET ZBYTE$ = CHR$(Ø): d.z = d.z + 1: PUT #
            2, d.z: d.z = d.z + 1: PUT #2, d.z: d.z = d
            .z + 1: PUT #2, d.z
[WZA] 1431: LSET ZBYTE$ = CHR$(255): d.z = d.z + 1: PUT
             #2, d.z: d.z = d.z + 1: PUT #2, d.z
[L1A] 1432: LSET ZBYTE$ = CHR$(Ø): d.z = d.z + 1: PUT #
            2, d.z: d.z = d.z + 1: PUT #2, d.z
[E2Ø] 1433: '
[3E3] 1434: 'Nun das SP-Register
[GS8] 1435: LSET ZBYTE$ = CHR$(stapel MOD 256): d.z = d
            .z + 1: PUT #2, d.z
[TW8] 1436: LSET ZBYTE$ = CHR$(INT(stapel / 255)): d.z
            = d.z + 1: PUT #2, d.z
[667] 1437: LSET ZBYTE$ = CHR$(237): d.z = d.z + 1: PUT
             #2, d.z
[YW6] 1438: LSET ZBYTE$ = CHR$(54): d.z = d.z + 1: PUT
            #2, d.z
[E2Ø] 1439: '
[5D3] 144Ø: 'Nun das IP-Register
[4L8] 1441: LSET ZBYTE$ = CHR$(code MOD 256): d.z = d.z
             + 1: PUT #2, d.z
[BW8] 1442: LSET ZBYTE$ = CHR$(INT(code / 255)): d.z =
            d.z + 1: PUT #2, d.z
[L1A] 1443: LSET ZBYTE$ = CHR$(Ø): d.z = d.z + 1: PUT #
            2, d.z: d.z = d.z + 1: PUT #2, d.z
[MW6] 1444: LSET ZBYTE$ = CHR$(3Ø): d.z = d.z + 1: PUT
            #2, d.z
[NHD] 1445: LSET ZBYTE$ = CHR$(Ø): d.z = d.z + 1: PUT #
            2, d.z: d.z = d.z + 1: PUT #2, d.z: d.z = d
            .z + 1: PUT #2, d.z
[KH6] 1446: LSET ZBYTE$ = CHR$(1): d.z = d.z + 1: PUT #
            2, d.z
[F13] 1447: LSET ZBYTE$ = CHR$(Ø)
[E2Ø] 1448: '
[R92] 1449: FOR i = 1 TO 483
[HE3] 145Ø:    d.z = d.z + 1: PUT #2, d.z
[U11] 1451: NEXT i
[E2Ø] 1452: '
[U13] 1453: FOR i = 1 TO laenge
[E81] 1454:    GET #3, i
[ZA1] 1455:    a$ = byte$
[HE2] 1456:    LSET ZBYTE$ = a$
[HE3] 1457:    d.z = d.z + 1: PUT #2, d.z
[U11] 1458: NEXT i
[E2Ø] 1459: '
[F51] 146Ø: CLOSE 3
[B51] 1461: CLOSE 2
[E2Ø] 1462: '
[651] 1463: RETURN
[E2Ø] 1464: '
[346] 1465: 'Initialisierung der flexiblen Daten
[AR1] 1466: flex.init:
[YE3] 1467: title$ = "": subttl$ = ""
[M11] 1468: d.z = Ø
[E2Ø] 1469: '
[KC8] 147Ø: operand = 1: konst2 = 4: regsum = 64: erwei
            bedarf = 128
[W3C] 1471: disp8bedarf = 256: disp16bedarf = 512: modb
            ytebedarf = 1Ø24: direkt8bedarf = 2Ø48
[ERB] 1472: direkt16bedarf = 4Ø96: richtung = 8192: spe
            icherbedarf = 16384: konst1 = &H8ØØØ
[TK9] 1473: akku8 = 1: akku16 = 2: reg8 = 4: reg16 = 8:
             speicherreg = 16: cs = 32
[3KA] 1474: segreg = 64: speicher = 128: direkt8 = 256:
             direkt16 = 512: nichts = 1Ø24
[Y67] 1475: zeichen = 2Ø48: nah = 4Ø96: weit = 8192: cl
            = 16384
[E2Ø] 1476: '
[5M2] 1477: stapelzeiger = Ø
[1R8] 1478: code = Ø: stapel = Ø: segmente = Ø: c2 = fa
            lsch: s2 = falsch
[DK3] 1479: fehler = Ø: warnung = Ø
[7D1] 148Ø: laenge = Ø
[E2Ø] 1481: '
[Z96] 1482: 'Öffnen der Ein- und Ausgabedateien
```

```
[WT4] 1483: OPEN quelldat$ FOR INPUT AS #1
[E2Ø] 1484: '
[V9G] 1485: IF asslis = 3 THEN l$ = LEFT$(zieldat$, LEN
            (zieldat$) - 3) + "lst": GOTO flex.oeffnen:
            'Ausgabe in eine Datei
[SBA] 1486: IF asslis = 2 THEN l$ = "lpt1:": GOTO flex.
            oeffnen:'Druckerausgabe
[EM1] 1487: l$ = "scrn:"
[E2Ø] 1488: '
[B62] 1489: flex.oeffnen:
[E2Ø] 149Ø: '
[DS3] 1491: OPEN l$ FOR OUTPUT AS #2
[2W1] 1492: O$ = zieldat$
[ØF3] 1493: PRINT : PRINT : PRINT
[E2Ø] 1494: '
[ZZ5] 1495: OPEN "work.wrk" FOR RANDOM AS #3 LEN = 1
[DT2] 1496: FIELD #3, 1 AS byte$
[BS1] 1497: LWIDTH = 79
[BS8] 1498: C$ = MID$(DATE$, 4, 2) + "." + LEFT$(DATE$,
             2) + "." + RIGHT$(DATE$, 4)
[DDB] 1499: IF asslis <> 4 THEN PRINT #2, quelldat$; TA
            B(LWIDTH - LEN(C$)); C$: PRINT #2,
[Y74] 15ØØ: IF asslis <> 4 THEN PRINT #2,
[RUD] 15Ø1: IF asslis <> 4 THEN PRINT #2, "Adr."; TAB(6
            ); "Hex-Code"; TAB(18); "Zeile"; TAB(25); "
            Quelltext"
[Y74] 15Ø2: IF asslis <> 4 THEN PRINT #2,
[E2Ø] 15Ø3: '
[651] 15Ø4: RETURN
[E2Ø] 15Ø5: '
[WL6] 15Ø6: 'Fehlerabfangroutine für das Laden der
[GN2] 15Ø7: ' Datei 'SAM3.DAT'
[912] 15Ø8: samdat.fehl:
[E2Ø] 15Ø9: '
[K9F] 151Ø: PRINT : PRINT : PRINT "   SAM III bricht ab
            , weil die Datei 'SAM3.DAT' nicht gefunden
            werden kann."
[E2Ø] 1511: '
[4MØ] 1512: END
[E2Ø] 1513: '
[XØ6] 1514: 'Fehlerabfangroutine für 'INCLUDE'
[T62] 1515: fehl.include:
[FS4] 1516: IF NOT DOS THEN GOTO fehl.inw1
[MHE] 1517: PRINT : PRINT "SAM III wurde durch einen Fe
            hler beim INCLUDE-Befehl unterbrochen.": PR
            INT
[4MØ] 1518: END
[E2Ø] 1519: '
[XM1] 152Ø: fehl.inw1:
[BPU] 1521: LOCATE 1Ø, 6: PRINT "Sie haben den INCLUDE-
            Befehl falsch benutzt und müssen Ihre Einga
            be": LOCATE 12, 14: PRINT "berichtigen. Drü
            cken Sie bitte eine beliebige Taste !"
[5H3] 1522: WHILE INKEY$ = "": WEND
[3WØ] 1523: CLOSE
[WS1] 1524: GOTO menue
[E2Ø] 1525: '
[YR5] 1526: 'Fehlerabfangroutine für 'MACRO'
[E42] 1527: macro.fehler:
[ZA4] 1528: IF NOT DOS THEN GOTO mac.w1
[F6E] 1529: PRINT : PRINT "SAM III wurde durch einen Fe
            hler beim MACRO-Befehl unterbrochen.": PRIN
            T
[4MØ] 153Ø: END
[E2Ø] 1531: '
[Ø61] 1532: mac.w1:
[ØUS] 1533: PRINT : PRINT "Sie haben den MACRO-Befehl f
            alsch benutzt und müssen Ihre Eingabe beric
            htigen.": PRINT : PRINT "Drücken Sie bitte
            eine beliebige Taste !"
[5H3] 1534: WHILE INKEY$ = "": WEND
[3WØ] 1535: CLOSE
[WS1] 1536: GOTO menue
[E2Ø] 1537: '
[R25] 1538: 'Fehlerabfangroutine für 'DO'
[WM1] 1539: do.fehler:
[344] 154Ø: IF NOT DOS THEN GOTO do.w1
[TGD] 1541: PRINT : PRINT "SAM III wurde durch einen Fe
            hler beim DO-Befehl unterbrochen.": PRINT
[4MØ] 1542: END
[E2Ø] 1543: '
[MWØ] 1544: do.w1:
[HØR] 1545: PRINT : PRINT "Der DO-Befehl wurde falsch b
            enutzt. Sie müssen den Befehl berichtigen."
            : PRINT : PRINT "Drücken Sie bitte eine bel
            iebige Taste !"
[5H3] 1546: WHILE INKEY$ = "": WEND
[3WØ] 1547: CLOSE
[WS1] 1548: GOTO menue
[E2Ø] 1549: '
[TU6] 155Ø: 'Erster Lauf (bereitet die Verwendung von
[4W6] 1551: ' Makros und deren Parameterübergabe vor)
[6R1] 1552: lauf.eins:
[ZM4] 1553: IF makros THEN GOSUB inc.exm
[HK5] 1554: OPEN "makros.wrk" FOR OUTPUT AS #6
[4E5] 1555: OPEN "work5.wrk" FOR OUTPUT AS #4
[E2Ø] 1556: '
[BE4] 1557: lauf = -1: aktsymbol = vordef
```

Listing 1. Die dritte SAM-Version »sam3.asc« läßt sich mit SBasic, Quick Basic und Turbo Basic compilieren

Sesam öffne dich!

BCi DisAsm$_{186}$
Intelligenter Disassembler Version 1.0

BCi DisAsm erschließt jedes Programm!

Wandelt den Hexdump in ein dokumentiertes Assemblerlisting um und kann nach Modifizierung oder Änderung durch einen Editor mit gängigen Assemblern oder Debuggern wieder assembliert werden.

BCi DisAsm – Es gibt kaum einen, der ihm das Wasser reichen kann!

Auch Experten kennen nicht immer die Lösung, aber sie wissen, wo sie suchen müssen.
BCi DisAsm hilft JEDEM, die verwirrenden und frustrierenden Software-Puzzle, welche sich beim Umgang mit dem Computer ergeben, zu lösen. Und zwar schneller als mit jedem anderen Produkt, das Sie erwerben können, egal welche Preisklasse Sie betrachten.
Mit dem BCi DisAsm erwerben Sie Know-How, das Sie sonst Jahre an Erfahrung kosten würde. Versuchen Sie nicht, das Rad neu zu erfinden!

BCi DisAsm – Der intelligente Disassembler
BCi DisAsm ist ein intelligenter, selbsttätig dokumentierender MS-DOS Disassembler. Die ausgefeilten Routinen im BCi DisAsm analysieren in kürzester Zeit selbst komplexen und umfangreichen Programmcode und zeigen dabei potentielle Problempunkte auf.

BCi DisAsm – Das Lernmittel
Sie brauchen kein Spezialist in Assembler-Programmierung zu sein. BCi DisAsm produziert aus jedem beliebigen Programm ihres IBM-kompatiblen Computers ein gut dokumentiertes, leicht verständliches Assemblerlisting.

BCi DisAsm – Einfach in der Handhabung
BCi DisAsm kann sowohl über Parameter von der DOS-Kommandozeile aus als auch interaktiv über Menüs gesteuert werden. Die Menüstruktur ist Lotus-kompatibel, selbstverständlich in Deutsch. Es besteht keine Notwendigkeit, neue Befehle zu lernen oder ständig im Handbuch nachzuschlagen. BCi DisAsm ist unmittelbar anwendbar!

Minimale Systemanforderungen
256 kB RAM, 8088/8086/80186/80286 oder 80386 CPU, MS-DOS oder PC-DOS Version 2.0 oder größer

BCi DisAsm wird Ihnen bei den folgenden Schritten helfen:

1. Lernen Sie Assembler-Programmierung, wenn Sie möchten.
2. Entdecken Sie, warum Programm XYZ auf ihrem Rechner nicht läuft, und tun Sie etwas dagegen.
3. Ändern oder entfernen Sie Programmteile.
4. Sichern Sie sich gegen destruktive Programme ab. Finden Sie einen "Virus", und entfernen Sie ihn.
5. Integrieren Sie Code aus compilierten Programmen in eigene Anwendungen.
6. Erhöhen Sie die Kompatibilität Ihrer Software.
7. Passen Sie Software an verschiedene MS-DOS-Versionen an.

© 1987 BCi, West Germany und USA
Lotus ist eingetragenes Warenzeichen der Lotus Corp.
MS-DOS ist eingetragenes Warenzeichen der Microsoft Corp.

BCi DisAsm kostet inkl. eines sehr ausführlichen deutschen Handbuchs (mehr als 100 Seiten) nur

Best.-Nr. 227 5 1/4"-Disk. *Best.-Nr. 233 3 1/2"-Disk.*

199,– DM unverbindliche Preisempfehlung
Wenn Sie über den DMV-Bestellservice bestellen, gilt folgendes:

Inland:		**Ausland:**	
Einzelpreis	199,– DM	Einzelpreis	199,– DM
zzgl. Versandkosten	4,– DM	zzgl. Versandkosten	6,– DM
Endpreis	203,– DM	Endpreis	205,– DM

Bestellen Sie noch heute mittels der entsprechenden Bestellkarte bei:

DMV-Verlag
Postfach 250 · 3440 Eschwege

Händleranfragen willkommen

```
[XU1] 1558: codezeig = Ø
[WV1] 1559: zeilnum = Ø
[E2Ø] 156Ø: '
[3E2] 1561: WHILE NOT EOF(1)
[2A6] 1562:   GOSUB lies.zeile:'Einlesen einer Zeile
[KM5] 1563:   GOSUB analyse:'Zeile übersetzen
[E2Ø] 1564:   '
[BR6] 1565:   'Falls Makro-Anweisungen vorhanden sind,
[G27] 1566:   ' so werden sie registriert und für die
[6S5] 1567:   ' spätere Verwendung vorbereitet
[KY5] 1568:   IF befehl$ <> "MACRO" THEN GOTO defp.ok
[TX2] 1569:   MACRONAME$ = marke$
[42H] 157Ø:   IF marke$ = "" THEN PRINT : PRINT "Das Ma
              kro ist nicht durch einen Namen gekennzeich
              net.": GOTO macro.fehler
[WZ7] 1571:   parameter$ = MID$(zeile$, INSTR(zeile$, b
              efehl$) + 6)
[E2Ø] 1572:   '
[ME4] 1573:   WHILE INSTR(parameter$, ";")
[ML8] 1574:     parameter$ = LEFT$(parameter$, INSTR(pa
              rameter$, ";") - 1)
[TRØ] 1575:   WEND
[E2Ø] 1576:   '
[1K4] 1577:   WHILE LEFT$(parameter$, 1) = " "
[LZ4] 1578:     parameter$ = MID$(parameter$, 2)
[TRØ] 1579:   WEND
[E2Ø] 158Ø:   '
[BU4] 1581:   WHILE RIGHT$(parameter$, 1) = " "
[6M7] 1582:     parameter$ = LEFT$(parameter$, LEN(para
              meter$) - 1)
[TRØ] 1583:   WEND
[E2Ø] 1584:   '
[R33] 1585:   werte = Ø: werte$ = ""
[262] 1586: param.falsch:
[Y2M] 1587:   IF werte = 9 THEN PRINT : PRINT "Zu viele
               oder falsche Parameter bei der Definition
              des Makros: "; marke$: GOTO macro.fehler
[9E6] 1588:   IF parameter$ = werte$ THEN GOTO mdef.anf
[P49] 1589:   IF werte = Ø THEN werte = 1: werte$ = "%1
              ": GOTO param.falsch
[8K2] 159Ø:   werte = werte + 1
[FP6] 1591:   werte$ = werte$ + ",%" + RIGHT$(STR$(wert
              e), 1)
[CW2] 1592:   GOTO param.falsch
[E2Ø] 1593:   '
[HE1] 1594: mdef.anf:
[FL5] 1595:   PRINT #6, "Makrodefinition: Anfang"
[ZE2] 1596:   PRINT #6, werte
[WD2] 1597:   PRINT #6, marke$
[E2Ø] 1598:   '
[LR1] 1599: enm.fehlt:
[BEF] 16ØØ:   IF EOF(1) THEN PRINT : PRINT "Der Befehl
              'ENDM' fehlt bei dem Makro: "; MACRONAME$:
              GOTO macro.fehler
[2A6] 16Ø1:   GOSUB lies.zeile:'Einlesen einer Zeile
[KM5] 16Ø2:   GOSUB analyse:'Zeile übersetzen
[E2Ø] 16Ø3:   '
[ØEB] 16Ø4:   IF befehl$ = "ENDM" THEN PRINT #6, "Makro
              definition: Ende": GOTO def.param
[5F2] 16Ø5:   PRINT #6, zeile$
[CE2] 16Ø6:   GOTO enm.fehlt
[E2Ø] 16Ø7:   '
[HP1] 16Ø8: def.param:
[ME1] 16Ø9:   zeile$ = ""
[E2Ø] 161Ø:   '
[882] 1611:   'Weitermachen
[EB1] 1612: defp.ok:
[5N2] 1613:   GOSUB aus.param
[285] 1614:   IF zeile$ <> "" THEN PRINT #4, zeile$
[TRØ] 1615: WEND
[E2Ø] 1616: '
[A14] 1617: CLOSE #4: CLOSE #6: CLOSE #1
[E2Ø] 1618: '
[651] 1619: RETURN
[E2Ø] 162Ø: '
[D36] 1621: 'Zweiter Include-Lauf zum Einbinden
[MS2] 1622: ' externer Makros
[PD1] 1623: inc.exm:
[3P8] 1624: codezeig = Ø: zeilnum = Ø: lauf = -2: aktsy
              mbol = vordef
[8E5] 1625: OPEN "work6.wrk" FOR OUTPUT AS #4
[E2Ø] 1626: '
[3E2] 1627: WHILE NOT EOF(1)
[8R2] 1628:   GOSUB lies.zeile
[T72] 1629:   GOSUB analyse
[BW6] 163Ø:   IF befehl$ <> "INCLUDE" THEN GOTO inc.sch
              luss
[277] 1631:   datei$ = MID$(zeile$, INSTR(zeile$, befeh
              l$) + 8)
[494] 1632:   WHILE INSTR(datei$, ";") <> Ø
[4B6] 1633:     datei$ = LEFT$(datei$, INSTR(datei$, ";
              ") - 1)
[TRØ] 1634:   WEND
[5T3] 1635:   WHILE LEFT$(datei$, 1) = " "
[3D3] 1636:     datei$ = MID$(datei$, 2)
[TRØ] 1637:   WEND
[X14] 1638:   WHILE RIGHT$(datei$, 1) = " "
[YC5] 1639:     datei$ = LEFT$(datei$, LEN(datei$) - 1)
[TRØ] 164Ø:   WEND
[R94] 1641:   OPEN datei$ FOR INPUT AS #5
[LE2] 1642:   WHILE NOT EOF(5)
[LE2] 1643:     LINE INPUT #5, a$
[GN1] 1644:     PRINT #4, a$
[TRØ] 1645:   WEND
[451] 1646:   CLOSE #5
[ME1] 1647:   zeile$ = ""
[U52] 1648: inc.schluss:
[5N2] 1649:   GOSUB aus.param
[285] 165Ø:   IF zeile$ <> "" THEN PRINT #4, zeile$
[TRØ] 1651: WEND
[E2Ø] 1652: '
[351] 1653: CLOSE #4
[Ø51] 1654: CLOSE #1
[E2Ø] 1655: '
[B15] 1656: OPEN "work6.wrk" FOR INPUT AS #1
[651] 1657: RETURN
```

Listing 1. Die dritte SAM-Version »sam3.asc« läßt sich mit SBasic, Quick Basic und Turbo Basic compilieren

```
[REØ]    1: 218
[MM4]    2: Anzahl der Assemblerbefehle
[3R1]    3: =,1,896,1Ø24,Ø
[A92]    4: AAA,55,1Ø24,1Ø24,1
[BF2]    5: AAD,213,1Ø24,1Ø24,1
[BG2]    6: AAM,212,1Ø24,1Ø24,1
[EB2]    7: AAS,63,1Ø24,1Ø24,1
[M52]    8: ADC,2Ø,3,768,41Ø1
[NH2]    9: ADC,128,12,768,4245
[V92]   1Ø: ADC,16,144,12,1Ø29
[AA2]   11: ADC,18,12,144,9221
[ØP2]   12: ADCB,128,144,256,4241
[BR2]   13: ADCW,129,144,768,4241
[P32]   14: ADD,4,3,768,41Ø1
[3K2]   15: ADD,128,12,768,4229
[R42]   16: ADD,Ø,144,12,1Ø29
[P52]   17: ADD,2,12,144,9221
[AP2]   18: ADDB,128,144,256,4225
[NR2]   19: ADDW,129,144,768,4225
[A72]   2Ø: AND,36,3,768,41Ø1
[DK2]   21: AND,128,12,768,4261
[NA2]   22: AND,32,144,12,1Ø29
[4B2]   23: AND,34,12,144,9221
[FR2]   24: ANDB,128,144,256,4257
[US2]   25: ANDW,129,144,768,4257
[DT2]   26: CALL,154,768,768,4Ø97
[2V2]   27: CALL,232,4Ø96,1Ø24,513
[WV2]   28: CALLF,255,144,1Ø24,153
[GW2]   29: CALLN,255,144,1Ø24,145
[4K2]   3Ø: CBW,152,1Ø24,1Ø24,1
[9H2]   31: CLC,248,1Ø24,1Ø24,1
[SG2]   32: CLD,252,1Ø24,1Ø24,1
[4H2]   33: CLI,25Ø,1Ø24,1Ø24,1
[ØH2]   34: CMC,245,1Ø24,1Ø24,1
[F82]   35: CMP,6Ø,3,768,41Ø1
[MM2]   36: CMP,128,12,768,4285
[PC2]   37: CMP,56,144,12,1Ø29
[5D2]   38: CMP,58,12,144,9221
[RS2]   39: CMPB,128,144,256,4281
[VS2]   4Ø: CMPSB,166,1Ø24,1Ø24,1
[7U2]   41: CMPSW,167,1Ø24,1Ø24,1
[2U2]   42: CMPW,129,144,768,4281
[VH2]   43: CWD,153,1Ø24,1Ø24,1
[S92]   44: DAA,39,1Ø24,1Ø24,1
[WB2]   45: DAS,47,1Ø24,1Ø24,1
[BC2]   46: DB,3,23Ø4,&HFFFF,Ø
[UA2]   47: DEC,254,4,1Ø24,137
[H22]   48: DEC,72,8,1Ø24,65
[SN2]   49: DECB,254,144,1Ø24,137
[EP2]   5Ø: DECW,255,144,1Ø24,137
[LE2]   51: DIV,246,3,144,8373
[1Ø2]   52: DS,4,768,128Ø,Ø
[M33]   53: END,9,&HFFFF,&HFFFF,Ø
[S73]   54: ENDP,6,&HFFFF,&HFFFF,Ø
[SF3]   55: ENDS,11,&HFFFF,&HFFFF,Ø
[Ø62]   56: EQU,1,896,1Ø24,Ø
[S83]   57: EVEN,8,&HFFFF,&HFFFF,Ø
[7L2]   58: HLT,244,1Ø24,1Ø24,1
[WK2]   59: IDIV,246,3,144,8381
[7N2]   6Ø: IMUL,246,3,144,8365
[8H1]   61: IN,236,1,8,1
[CH1]   62: IN,237,2,8,1
[E42]   63: IN,228,3,256,2Ø53
[3C2]   64: INC,254,4,1Ø24,129
[U32]   65: INC,64,8,1Ø24,65
[ØR2]   66: INCB,254,144,1Ø24,129
[PR2]   67: INCW,255,144,1Ø24,129
[3L2]   68: INT,2Ø4,1Ø24,1Ø24,1
[NT2]   69: INT,2Ø5,256,1Ø24,2Ø49
[XM2]   7Ø: INTO,2Ø6,1Ø24,1Ø24,1
[LL2]   71: IRET,2Ø7,1Ø24,1Ø24,1
[ZG2]   72: JA,119,4Ø96,1Ø24,257
[KP2]   73: JAE,115,4Ø96,1Ø24,257
```

```
[LG2]    74: JB,114,4Ø96,1Ø24,257
[1R2]    75: JBE,118,4Ø96,1Ø24,257
[6Y2]    76: JCXZ,227,4Ø96,1Ø24,257
[2H2]    77: JE,116,4Ø96,1Ø24,257
[DH2]    78: JG,127,4Ø96,1Ø24,257
[7R2]    79: JGE,125,4Ø96,1Ø24,257
[LH2]    8Ø: JL,124,4Ø96,1Ø24,257
[UR2]    81: JLE,126,4Ø96,1Ø24,257
[BT2]    82: JMP,234,768,768,4Ø97
[2T2]    83: JMP,233,4Ø96,1Ø24,513
[WS2]    84: JMPF,255,144,1Ø24,169
[MS2]    85: JMPN,255,144,1Ø24,161
[ZX2]    86: JMPS,235,4Ø96,1Ø24,257
[1S2]    87: JNE,117,4Ø96,1Ø24,257
[TS2]    88: JNO,113,4Ø96,1Ø24,257
[ØT2]    89: JNP,123,4Ø96,1Ø24,257
[4T2]    9Ø: JNS,121,4Ø96,1Ø24,257
[NU2]    91: JNZ,117,4Ø96,1Ø24,257
[MH2]    92: JO,112,4Ø96,1Ø24,257
[TH2]    93: JP,122,4Ø96,1Ø24,257
[PR2]    94: JPE,122,4Ø96,1Ø24,257
[2T2]    95: JPO,123,4Ø96,1Ø24,257
[WH2]    96: JS,12Ø,4Ø96,1Ø24,257
[1L2]    97: JZ,116,4Ø96,1Ø24,257
[RK2]    98: LAHF,159,1Ø24,1Ø24,1
[6F2]    99: LDS,197,8,144,9217
[UB2]   1ØØ: LEA,141,8,144,9217
[5F2]   1Ø1: LES,196,8,144,9217
[8L2]   1Ø2: LOCK,24Ø,1Ø24,1Ø24,1
[WR2]   1Ø3: LODSB,172,1Ø24,1Ø24,1
[8T2]   1Ø4: LODSW,173,1Ø24,1Ø24,1
[ZX2]   1Ø5: LOOP,226,4Ø96,1Ø24,257
[D63]   1Ø6: LOOPE,225,4Ø96,1Ø24,257
[S93]   1Ø7: LOOPNE,224,4Ø96,1Ø24,257
[RB3]   1Ø8: LOOPNZ,224,4Ø96,1Ø24,257
[R73]   1Ø9: LOOPZ,225,4Ø96,1Ø24,257
[VN2]   11Ø: MOV,16Ø,3,128,16389
[XP2]   111: MOV,162,128,3,16389
[PM2]   112: MOV,138,12,144,9221
[RN2]   113: MOV,136,144,12,1Ø29
[BG2]   114: MOV,184,8,768,4161
[FF2]   115: MOV,176,4,256,4161
[6N2]   116: MOV,142,64,144,9217
[EN2]   117: MOV,14Ø,144,96,1Ø25
[7V2]   118: MOVB,198,144,256,4225
[9U2]   119: MOVSB,164,1Ø24,1Ø24,1
[NV2]   12Ø: MOVSW,165,1Ø24,1Ø24,1
[KW2]   121: MOVW,199,144,768,4225
[9F2]   122: MUL,246,3,144,8357
[ZK2]   123: NEG,246,12,1Ø24,157
[MP2]   124: NEGB,246,144,1Ø24,153
[AR2]   125: NEGW,247,144,1Ø24,153
[AL2]   126: NOP,144,1Ø24,1Ø24,1
[PN2]   127: NOT,246,12,1Ø24,149
[9T2]   128: NOTB,246,144,1Ø24,145
[ZT2]   129: NOTW,247,144,1Ø24,145
[X22]   13Ø: OR,12,3,768,41Ø1
[XA2]   131: OR,128,12,768,4237
[X12]   132: OR,8,144,12,1Ø29
[X42]   133: OR,1Ø,12,144,9221
[AP2]   134: ORB,128,144,256,4233
[Y42]   135: ORG,2,768,1Ø24,Ø
[RT2]   136: ORW,129,144,768,4233
[8V1]   137: OUT,238,8,1,1
[GV1]   138: OUT,239,8,2,1
[EF2]   139: OUT,23Ø,256,3,2Ø53
[F62]   14Ø: POP,88,8,1Ø24,65
[3P2]   141: POP,143,144,1Ø24,129
[D62]   142: POP,7,64,1Ø24,65
[7N2]   143: POPF,157,1Ø24,1Ø24,1
[8M2]   144: PROC,5,12288,1Ø24,Ø
[RA2]   145: PUSH,8Ø,8,1Ø24,65
[BU2]   146: PUSH,255,144,1Ø24,177
[AA2]   147: PUSH,6,96,1Ø24,65
[NU2]   148: PUSHF,156,1Ø24,1Ø24,1
[553]   149: RCL,2Ø8,12,174Ø8,&H8Ø95
[Ø93]   15Ø: RCLB,2Ø8,144,174Ø8,&H8Ø91
[P93]   151: RCLW,2Ø9,144,174Ø8,&H8Ø91
[T73]   152: RCR,2Ø8,12,174Ø8,&H8Ø9D
[8A3]   153: RCRB,2Ø8,144,174Ø8,&H8Ø99
[YA3]   154: RCRW,2Ø9,144,174Ø8,&H8Ø99
[KK2]   155: REP,243,1Ø24,1Ø24,1
[7M2]   156: REPE,243,1Ø24,1Ø24,1
[WS2]   157: REPNE,242,1Ø24,1Ø24,1
[6U2]   158: REPNZ,242,1Ø24,1Ø24,1
[WM2]   159: REPZ,243,1Ø24,1Ø24,1
[KL2]   16Ø: RET,2Ø3,1Ø24,8192,1
[BM2]   161: RET,195,1Ø24,4Ø96,1
[7V2]   162: RET,2Ø2,768,8192,4Ø97
[1W2]   163: RET,194,768,4Ø96,4Ø97
[663]   164: ROL,2Ø8,12,174Ø8,&H8Ø85
[3A3]   165: ROLB,2Ø8,144,174Ø8,&H8Ø81
[TA3]   166: ROLW,2Ø9,144,174Ø8,&H8Ø81
[U83]   167: ROR,2Ø8,12,174Ø8,&H8Ø8D
[BB3]   168: RORB,2Ø8,144,174Ø8,&H8Ø89
[1C3]   169: RORW,2Ø9,144,174Ø8,&H8Ø89
```

Listing 2. Die Datei »sam3.dat« enthält die Daten, die SAM III für die Arbeit benötigt

DOS INTERNATIONAL

Tips & Tricks gesucht!

Die DOS International ist eine Zeitschrift zum Mitmachen. Vor allem die Tips & Tricks-Rubriken leben von Ihren Beiträgen, Ihren Ideen und Programmen. Sie müssen kein Schriftsteller sein, damit ein Beitrag aus Ihrer Feder (beziehungsweise Ihrer Tastatur) veröffentlicht wird. Einschicken können Sie alles, was irgendwie mit der Programmierung und Bedienung des PC zusammenhängt. Das sind zum Beispiel wichtige Hinweise zu Anwendungs-Software, trickreiche Stapeldateien, interessante Basic-, Assembler-, C-, oder Pascal-Routinen, aber auch Tips zur PC-Hardware. Für jeden veröffentlichten Trick gibt es – je nach Länge und Qualität – ein angemessenes Honorar.

Bitte beachten Sie beim Vorbereiten einer Einsendung die folgenden Hinweise, damit der Beitrag in der Redaktion schnell und unproblematisch bearbeitet werden kann:

- Legen Sie eine Diskette bei mit allen zum Trick gehörigen Dateien (Quelltext, ausführbares Programm, Beschreibung).
- Das Anschreiben sollte in kurzen Stichpunkten Ihre Idee beschreiben.
- Diskette und Anschreiben versehen Sie bitte mit Ihrer genauen Adresse und mit dem Programmnamen.
- Sie ersparen uns viel Arbeit, wenn Sie Quelltexte auf 44 Zeichen pro Zeile formatieren. Steuerzeichen sollten nicht enthalten sein.
- Die Programmbeschreibung sollte als unformatierter ASCII-Text vorliegen.

Spätestens zwei Wochen, nachdem Ihre Einsendung bei uns eingegangen ist, erhalten Sie schriftlich Bescheid.

Unsere Adresse:

Redaktionsbüro
Everts & Hagedorn
Redaktion DOS International
Stichwort Tips & Tricks
Hans-Stießberger-Straße 3
8013 Haar/München

Wir freuen uns über jede Zuschrift.

```
[4L2]   17Ø: SAHF,158,1Ø24,1Ø24,1
[G83]   171: SAR,2Ø8,12,174Ø8,&H8ØBD
[DA3]   172: SARB,2Ø8,144,174Ø8,&H8ØB9
[3B3]   173: SARW,2Ø9,144,174Ø8,&H8ØB9
[672]   174: SBB,28,3,768,41Ø1
[AK2]   175: SBB,128,12,768,4253
[HA2]   176: SBB,24,144,12,1Ø29
[ØB2]   177: SBB,26,12,144,9221
[AR2]   178: SBBB,128,144,256,4249
[NS2]   179: SBBW,129,144,768,4249
[YP2]   18Ø: SCASB,174,1Ø24,1Ø24,1
[AS2]   181: SCASW,175,1Ø24,1Ø24,1
[872]   182: SEG,38,96,1Ø24,65
[AZ3]   183: SEGMENT,1Ø,&HFFFF,&HFFFF,Ø
[E63]   184: SHL,2Ø8,12,174Ø8,&H8ØA5
[T93]   185: SHLB,2Ø8,144,174Ø8,&H8ØA1
[FA3]   186: SHLW,2Ø9,144,174Ø8,&H8ØA1
[293]   187: SHR,2Ø8,12,174Ø8,&H8ØAD
[1B3]   188: SHRB,2Ø8,144,174Ø8,&H8ØA9
[RB3]   189: SHRW,2Ø9,144,174Ø8,&H8ØA9
[5L2]   19Ø: STC,249,1Ø24,1Ø24,1
[MK2]   191: STD,253,1Ø24,1Ø24,1
[ØL2]   192: STI,251,1Ø24,1Ø24,1
[1U2]   193: STOSB,17Ø,1Ø24,1Ø24,1
[DV2]   194: STOSW,171,1Ø24,1Ø24,1
[R82]   195: SUB,44,3,768,41Ø1
[YM2]   196: SUB,128,12,768,4269
[2C2]   197: SUB,4Ø,144,12,1Ø29
[HC2]   198: SUB,42,12,144,9221
[XS2]   199: SUBB,128,144,256,4265
[6S3]   2ØØ: SUBTTL,7,&HFFFF,&HFFFF,Ø
[8U2]   2Ø1: SUBW,129,144,768,4265
[SM2]   2Ø2: TEST,168,3,768,41Ø1
[5S2]   2Ø3: TEST,246,12,768,4229
[1P2]   2Ø4: TEST,132,144,12,1Ø29
[UP2]   2Ø5: TEST,132,12,144,9221
[1Z2]   2Ø6: TESTB,246,144,256,4225
[613]   2Ø7: TESTW,247,144,768,4225
[VL3]   2Ø8: TITLE,7,&HFFFF,&HFFFF,Ø
[VL2]   2Ø9: WAIT,155,1Ø24,1Ø24,1
[DD2]   21Ø: XCHG,144,8,1Ø24,65
[G82]   211: XCHG,144,2,8,8257
[CM2]   212: XCHG,134,144,12,1Ø29
[5N2]   213: XCHG,134,12,144,9221
[HL2]   214: XLAT,215,1Ø24,1Ø24,1
[BA2]   215: XOR,52,3,768,41Ø1
[KE2]   216: XOR,48,144,12,1Ø29
[5E2]   217: XOR,5Ø,12,144,9221
[HP2]   218: XOR,128,12,768,4277
[KU2]   219: XORB,128,144,256,4273
[XV2]   22Ø: XORW,129,144,768,4273
[4RØ]   221: 29,5ØØ
[TBA]   222: Anzahl der vordefinierten Symbole, maximale
             Anzahl der Symbole
[TG1]   223: AX,192,Ø,26
[ØH1]   224: CX,193,8,24
[CK1]   225: DX,194,16,24
[8K1]   226: BX,195,24,24
[KK1]   227: SP,196,32,24
[GH1]   228: BP,197,4Ø,24
[GK1]   229: SI,198,48,24
[KH1]   23Ø: DI,199,56,24
[1F1]   231: AL,192,Ø,21
[HT1]   232: CL,193,8,164Ø4
[4H1]   233: DL,194,16,2Ø
[ØH1]   234: BL,195,24,2Ø
[KG1]   235: AH,196,32,2Ø
[PG1]   236: CH,197,4Ø,2Ø
[BH1]   237: DH,198,48,2Ø
[7H1]   238: BH,199,56,2Ø
[W72]   239: [BX+SI],Ø,Ø,16
[Ø72]   24Ø: [BX+DI],1,Ø,16
[Ø72]   241: [BP+SI],2,Ø,16
[462]   242: [BP+DI],3,Ø,16
[7P1]   243: [SI],4,Ø,16
[XM1]   244: [DI],5,Ø,16
[TP1]   245: [BX],7,Ø,16
[T41]   246: ES,Ø,Ø,64
[651]   247: CS,Ø,8,32
[PB1]   248: SS,Ø,16,64
[LA1]   249: DS,Ø,24,64
[2R1]   25Ø: NEAR,Ø,Ø,4Ø96
[CN1]   251: FAR,Ø,Ø,8192
```

Listing 2. Die Datei »sam3.dat« enthält die Daten, die SAM III für die Arbeit benötigt

```
[8R6]    1: ;Dieses Programm gibt auf dem Bildschirm
[GP6]    2: ; eine Zeichenkette aus und kehrt darauf
[W96]    3: ; in die MS-DOS-Kommandoebene zurück
[ØØØ]    4:
[KC6]    5: ;Programm startet ab 1ØØhex (Com-Datei)
[WP1]    6:          org       1ØØh
[FS4]    7: ;Sprung zum Programmanfang
[RC3]    8: start    jmps      anfang
```

```
[ØØØ]    9:
[562]   1Ø: ;Datenbereich
[V62]   11: text     db 13,1Ø
[7U5]   12:          db 'Ein Beispielprogramm für den '
[MM4]   13:          db 'neuen SAM III aus der DOS '
[W44]   14:          db 'International.',13,1Ø,'$'
[ØØØ]   15:
[HU2]   16: ;Programmbereich
[Z84]   17: ;Adresse der Zeichenkette
[4N4]   18: anfang   mov  dx,offset(text)
[Ø76]   19: ;Zeichenkette über Interrupt 21hex,
[Z12]   2Ø:          mov       ah,Ø9h
[HD7]   21: ;Funktion Ø9hex auf den Bildschirm ausgeben
[NK1]   22:          int       21h
[SG5]   23: ;Programm über Interrupt 21hex,
[L32]   24:          mov       ah,4ch
[114]   25: ;Funktion 4chex beenden
[NK1]   26:          int       21h
[1A2]   27: ;Programmende
[4MØ]   28:          end
```

Listing 3. Das Demoprogramm »ausgabe.sam« zeigt, wie Sie einen Text auf den Bildschirm ausgeben

```
[7M2]    1: sam3 %1.sam %1.exe
[CL1]    2: exe2bin %1
[ØG1]    3: del %1.exe
[4D2]    4: ren %1.bin %1.com
```

Listing 4. Die Stapeldatei »sam.bat« erleichtert nochmals den Umgang mit SAM III

```
[TE6]    1: prog   segment code  ;Codesegmentanfang
[854]    2:        mov ah,2      ;Die Buchstaben
[MG2]    3:        mov dl,65     ; 'A'
[LN2]    4:        int 21h       ; und
[TG2]    5:        mov dl,66     ; 'B'
[LN2]    6:        int 21h       ; und
[YG2]    7:        mov dl,67     ; 'C'
[LS4]    8:        int 21h       ;nacheinander auf
[8C4]    9:        mov ah,4ch    ;den Bildschirm
[ZD3]   1Ø:        int 21h       ;ausgeben
[UA5]   11: prog   ends          ;Codesegmentende
[P3Ø]   12: ;
[GY6]   13: stapel segment stack ;Stapelsegmentanfang
[2B1]   14:        db  Ø,Ø,Ø,Ø
[326]   15: stapel ends          ;Stapelsegmentende
[P3Ø]   16: ;
[XU3]   17:        end           ;Programmende
```

Listing 5. Das Programmbeispiel »abcdemo.sam« arbeitet mit einem Code- und einem Stapelsegment

```
[TE6]    1: prog   segment code  ;Codesegmentanfang
[P3Ø]    2: ;
[8R6]    3:      include inc.sam ;Include-Datei einbind
               en
[P3Ø]    4: ;
[UA5]    5: prog   ends          ;Codesegmentende
```

```
[P3Ø]    6: ;
[GY6]    7: stapel segment stack ;Stapelsegmentanfang
[2B1]    8:         db  Ø,Ø,Ø,Ø
[326]    9: stapel ends           ;Stapelsegmentende
[P3Ø]   1Ø: ;
[XU3]   11:         end           ;Programmende
```

Listing 6. Das Programm »incdemo.sam« bindet eine Include-Datei ein

```
[854]    1:         mov ah,2      ;Die Buchstaben
[MG2]    2:         mov dl,65     ; 'A'
[LN2]    3:         int 21h       ; und
[TG2]    4:         mov dl,66     ; 'B'
[LN2]    5:         int 21h       ; und
[YG2]    6:         mov dl,67     ; 'C'
[LS4]    7:         int 21h       ;nacheinander auf
[8C4]    8:         mov ah,4ch    ;den Bildschirm
[ZD3]    9:         int 21h       ;ausgeben
```

Listing 7. Die Include-Datei »inc.sam« für Listing 6

```
[ZW6]    1: prog    segment code     ;Codesegmentanfang
[P3Ø]    2: ;
[AB6]    3: ausgabe macro %1         ;Anfang der ersten
[B15]    4:         mov   ah,Ø2h     ; Makrodefinition
[6M1]    5:         mov   dl,%1
[N81]    6:         int   21h
[K84]    7:         endm             ;Ende der ersten
[YX4]    8: ;                        ; Makrodefinition
[5H6]    9: schluss macro            ;Anfang der zweite
            n
[V35]   1Ø:         mov   ah,4ch     ; Makrodefinition
[N81]   11:         int   21h
[HPØ]   12:         endm
[P3Ø]   13: ;
[3T4]   14:         do    ausgabe,65 ;Die Buchstaben
[HM4]   15:         do    ausgabe,66 ; 'A', 'B' und 'C'
[7E4]   16:         do    ausgabe,67 ; auf den Bild-
[NU2]   17:                          ; schirm ausgeben
[P3Ø]   18: ;
[MM4]   19:         do    schluss    ;Rückkehr ins
[5M4]   2Ø: ;                        ; Hauptprogramm
[SL5]   21: prog    ends             ;Codesegmentende
[P3Ø]   22: ;
[XØ4]   23:         end              ;Programmende
```

Listing 8. »makro.sam« zeigt ein Beispiel für die Verwendung von Makros

```
[8R6]    1: ;Dieses Programm gibt auf dem Bildschirm
[9A6]    2: ; eine Zeichenkette aus und liest eine
[ØZ2]    3: ; Zeichenkette ein
[ØØØ]    4:
[KC6]    5: ;Programm startet ab 1ØØhex (Com-Datei)
[WP1]    6:         org       1ØØh
[FS4]    7: ;Sprung zum Programmanfang
[RC3]    8: start    jmps      anfang
[ØØØ]    9:
[562]   1Ø: ;Datenbereich
[V62]   11: text    db 13,1Ø
[F76]   12:         db 'Bitte geben Sie Ihren Namen ei
            n: $'
[4Z7]   13: ;Erstes Byte des Eingabepuffers (Pufferläng
            e)
[3A2]   14: max     db        2Ø
[KE2]   15: ;Eingabepuffer
[HP2]   16:         db        '                    '
[ØØØ]   17:
[HU2]   18: ;Programmbereich
[Z84]   19: ;Adresse der Zeichenkette
[4N4]   2Ø: anfang  mov  dx,offset(text)
[Ø76]   21: ;Zeichenkette über Interrupt 21hex,
[Z12]   22:         mov       ah,Ø9h
[HD7]   23: ;Funktion Ø9hex auf den Bildschirm ausgeben
[NK1]   24:         int       21h
```

```
[2F3]   25: ;Adresse des Puffers
[L83]   26:         mov  dx,offset(max)
[CU3]   27: ;Zeichenkette einlesen
[Z22]   28:         mov       ah,Øah
[NK1]   29:         int       21h
[SG5]   3Ø: ;Programm über Interrupt 21hex,
[L32]   31:         mov       ah,4ch
[114]   32: ;Funktion 4chex beenden
[NK1]   33:         int       21h
[1A2]   34: ;Programmende
[4MØ]   35:         end
```

Listing 9. Die Routine »eingabe1.sam« dient zur Eingabe von Daten

```
[D27]    1: ;Dieses Programm gibt über ein Makro eine
[3G6]    2: ; Zeichenkette auf dem Bildschirm aus und
[VV4]    3: ; liest eine Zeichenkette ein
[ØØØ]    4:
[7W2]    5: ;Makrodefinition
[BD3]    6: funktion macro     %1,%2
[WW2]    7:         mov  dx,offset(%2)
[PX1]    8:         mov       ah,%1
[NK1]    9:         int       21h
[HPØ]   1Ø:         endm
[ØØØ]   11:
[KC6]   12: ;Programm startet ab 1ØØhex (Com-Datei)
[WP1]   13:         org       1ØØh
[FS4]   14: ;Sprung zum Programmanfang
[RC3]   15: start   jmps      anfang
[ØØØ]   16:
[562]   17: ;Datenbereich
[V62]   18: text    db 13,1Ø
[F76]   19:         db 'Bitte geben Sie Ihren Namen ei
            n: $'
[4Z7]   2Ø: ;Erstes Byte des Eingabepuffers (Pufferläng
            e)
[3A2]   21: max     db        2Ø
[KE2]   22: ;Eingabepuffer
[HP2]   23:         db        '                    '
[ØØØ]   24:
[HU2]   25: ;Programmbereich
[KT3]   26: ;Zeichenkette ausgeben
[NP4]   27: anfang  do funktion,Ø9h,text
[CU3]   28: ;Zeichenkette einlesen
[YA3]   29:         do funktion,Øah,max
[SG5]   3Ø: ;Programm über Interrupt 21hex,
[L32]   31:         mov       ah,4ch
[114]   32: ;Funktion 4chex beenden
[NK1]   33:         int       21h
[1A2]   34: ;Programmende
[4MØ]   35:         end
```

Listing 10. »eingabe2.sam« demonstriert die Makroversion der Eingaberoutine

```
[TE6]    1: prog    segment code  ;Codesegmentanfang
[P3Ø]    2: ;
[ØK1]    3:         call proz
[3N4]    4:         mov  ah,4ch   ;Rückkehr ins
[FA4]    5:         int  21h      ; Hauptprogramm
[P3Ø]    6: ;
[G96]    7: proz    proc near     ;Anfang der Prozedur
[Z64]    8:         mov  ah,2     ;Die Buchstaben
[XL2]    9:         mov  dl,68    ; 'D'
[YK2]   1Ø:         int  21h      ; und
[3N2]   11:         mov  dl,79    ; 'O'
[YK2]   12:         int  21h      ; und
[ZM2]   13:         mov  dl,83    ; 'S'
[ØC3]   14:         int  21h      ; ausgeben
[F33]   15:         ret           ;Ende der
[893]   16:         endp          ; Prozedur
[P3Ø]   17: ;
[UA5]   18: prog    ends          ;Codesegmentende
[P3Ø]   19: ;
[GY6]   2Ø: stapel segment stack ;Stapelsegmentanfang
[2B1]   21:         db  Ø,Ø,Ø,Ø
[326]   22: stapel ends           ;Stapelsegmentende
[P3Ø]   23: ;
[XU3]   24:         end           ;Programmende
```

Listing 11. Daß SAM III auch Prozeduren keine Schwierigkeiten bereiten, beweist das Beispiel »prozedur.sam«

Impressum

Herausgeber
Christian Widuch

Leitung des Redaktionsbüros
Volker Everts (ev), Andreas Hagedorn (hg)

Chefredakteur
Martin Aschoff (ma)

Stellvertretende Chefredakteurin
Eva-Maria Hierlmeyer (hi)

Redaktion
Alois Kneisle (al), Manfred Kohlen (mk), Werner Niedermaier (wn), Thomas Röder (tr), Herwig Weihe (hw), Stefan Willmeroth (sw), Johann Zieger (zi)

Schlußredaktion
Barbara Mooser, Karl-Heinz Peter, Markus Stein, Andrea Voß

Redaktionsassistenz
Claudia Beer, Dorothea Lehner

Koordination
Stefan Ritter

Produktionsleitung
Gerd Köberich

Layout, Fotosatz, Fotografie, Lektorat, Montage, Werbegestaltung
DMV-Verlag

Anzeigenverkaufsleitung
Wolfgang Schnell

Anzeigenverkauf für PLZ 1,4,5
Gerlinde Rachow (0 56 51) 80 09 - 53
Wolfgang Brill (0 56 51) 80 09 - 51

Anzeigenverkauf für PLZ 2 + 3
DMV-Verlagsbüro Hamburg
Ohlsdorfer Straße 34, 2000 Hamburg 60
Sylvia Ehrenpfordt, Tel. (040) 46 12 33

Anzeigenverkauf für PLZ 6 – 8
DMV-Verlagsbüro München
Zaunkönigweg 2c, 8000 München 82,
Telefon: (0 89) 4 39 10 87, Telefax: (0 89) 4 39 10 80
Leitung: Britta Fiebig
Anzeigenverkauf: Monika Schöbel, Jens Dhein

Anzeigenverwaltung und Disposition
Andrea Giese, Karina Ehrlich, Beate Kranz

Anschrift Redaktion:
Redaktionsbüro Everts & Hagedorn
Redaktion DOS International,
Hans-Stießberger-Str. 3, 8013 Haar/München
Telefon: (0 89) 46 10 01 -0
Telefax: (0 89) 46 10 01-90

Anschrift Verlag:
DMV Daten und Medien Verlag
Widuch GmbH & Co. KG
Fuldaer Straße 6, 3440 Eschwege
Telefon: (0 56 51) 80 09-0
Telefax: (0 56 51) 80 09-33

Vertrieb
Verlagsunion Erich Pabel-Arthur Moewig KG (VPM)
Friedrich-Bergius-Straße 20
6200 Wiesbaden

Druck
Druckerei Jungfer, 3420 Herzberg

Bezugspreise
DOS Extra erscheint viermal jährlich.
Einzelpreis DM 18,–, 145 öS, 18,– sfr.

Abonnementpreise
Die Preise verstehen sich grundsätzlich einschließlich Porto und Verpackung.

Inland: Bezugspreis 4 Ausgaben: DM 60,–

Europäisches Ausland: Bezugspreis 4 Ausgaben: DM 70,–

Außereuropäisches Ausland: Bezugspreis 4 Ausgaben: DM 80,–

Bankverbindungen:
Postscheck Frankfurt/M: Kto.-Nr.: 23043-608
Raiffeisenbank Eschwege:
BLZ: 522 603 85, Kto.-Nr.: 245 7008

Die Abonnementbestellung kann innerhalb einer Woche nach Auftrag beim DMV-Verlag, Postfach 250, 3440 Eschwege, schriftlich widerrufen werden. Zur Wahrung der Frist reicht der Poststempel. Das Abonnement verlängert sich automatisch um 4 Ausgaben, wenn es nicht mindestens 6 Wochen vor Ablauf beim Verlag schriftlich gekündigt wird.

Manuskripte, Programme:
bitte an das Redaktionsbüro Everts & Hagedorn, Redaktion DOS International, Hans-Stießberger-Straße 3, 8013 Haar/München senden.

Für unverlangt eingesandte Manuskripte und Datenträger sowie Fotos übernehmen der Verlag und die Redaktion keine Haftung. Die Zustimmung zum Abdruck wird vorausgesetzt.
Das Urheberrecht für veröffentlichte Manuskripte liegt ausschließlich beim Verlag. Nachdruck sowie Vervielfältigung oder sonstige Verwertung von Texten nur mit schriftlicher Genehmigung des Verlages. Für veröffentlichte Programme und Basteleien übernehmen wir weder Gewähr noch Haftung. Schaltungen und verwendete Bezeichnungen müssen nicht frei von gewerblichen Schutzrechten sein. Namentlich gekennzeichnete Fremdbeiträge geben nicht in jedem Fall die Meinung der Redaktion wieder.

TOOLBOX-Spezial

Reichhaltige Software zum Superpreis, mit Dokumentation auf der Diskette.

Spezial I: Turbo Prolog
Eine Toolbox mit Routinen zur "Mustererkennung", je einem Programm für statistische Auswertungen und analytisches Integrieren und Differenzieren mathematischer Terme, dBase-Dateien als Prolog-Wissensbasis und ein Turbo-Prolog-Tracer. Dokumentationen zu den Programmen auf der Diskette.

Spezial II: PasComp – Standard Pascal-Compiler
Kompletter Quellcode des Pascal-Compilers und des P-Code-Interpreters für Turbo Pascal 4.0 und CCD-Atari ST+. Dazu jeweils die lauffähigen Compilate und jetzt auch alle Texte der PASCAL-Serie PASCOMP als Dokumentation auf der Diskette!

Spezial III: Echtzeitverarbeitung
Komplettes Echtzeitsystem zu Studienzwecken in Turbo Pascal 3.0 Quellcode: Scheduler, Debugger, Utilities und Beispiele. Anpassungshinweise an andere Compiler, Programmiertechniken und Grundlagen der Echtzeitverarbeitung als Dokumentation auf der Diskette. Auch mit Turbo Pascal 4/5 kompilierbar.

Spezial IV: CALC & Kurvendiskussion
Mit dem Mathe-Compiler CALC können Formeln und Funktionen zur Auswertung direkt über die Tastatur eingegeben werden. Jetzt auch Differenzieren! Lauffähiges Demonstrationsprogramm "Kurvendiskussion" mit Bildschirmmodulen für Apple II, IBM-CGA und EGA. Komplette Quellcodes in Turbo Pascal 3.0 und jetzt auch als Turbo 4.0 Unit. Mit Dokumentation auf Diskette.

Spezial V: STRUKTO Plus "Pascal"
Erzeugung von Strukturdateien, grafischen Struktogrammen und Programmorganisationsplänen (POP) aus Pascal-Quelltexten oder Strukturdateien. Ausgabe wahlweise auf Drucker, Bildschirm oder in Dateien, inklusive einer Fehleranalyse. Turbo Pascal 3.0 Quellcode, problemlos auch für Turbo Pascal 4/5 verwendbar. Anleitung und Dokumentation auf der Diskette.

Spezial VI: Spiel & Spaß
Die besten Spiele und Knobels aus PASCAL und TOOLBOX jetzt in Farbe: Stratego, Schiebung, Inverso, der Zahlenwurm... Dazu ein kompletter Sprite-Editor für eigene Ausgestaltungen und eine Turbo Pascal "Spiele-Unit" mit vielen grundlegenden Funktionen. Alles als Turbo Pascal 4.0 Quellcode und als lauffähige Programme. Anleitungen als Datenfile auf Diskette.

Spezial VII: Deutsche Silbentrennung
Tabellengesteuerte deutsche Silbentrennung mit mehr als 99% Treffsicherheit: Dokumentierte Algorithmen und Turbo Pascal 3.0 (auch 4/5) Quellcode. Ein lauffähiges Programm fügt beliebige Trennzeichen in einen ASCII-Text ein. Dazu passend ein System zum Aufbau beliebig großer Wörterbücher. Quellcode in Modula-2 und demonstrierte Anwendungen. Dokumentation auf Diskette.

Spezial VIII: Spieltheorie
Grafische Lösung von 2*n Matrix-Spielen (TP ab 3.0), Simplex-Methode (TP ab 3.0), universelles "Minimax-Modul" (TP ab 3.0, Quick Basic 4.5), Beispielprogramm "Vier gewinnt" (TP ab 3.0), Programm zur Lösung von Schachproblemen (TP ab 3.0) und "Kalaha-Awari" (TP 4/5).

Spezial IX: HPGL-Plotter
Treiber-Toolbox für Turbo Pascal 4/5 und Turbo Basic für die parallele Schnittstelle oder eine Druckdatei und je ein Programm zum Zeichnen von zwei- und dreidimensionalen Funktionen (TP 4/5). Dazu eine Utility zur Ausgabe eines Hardware-Plotter-Zeichensatzes und eine Turbo Pascal UNIT zur Textausgabe mit BGI-Zeichensätzen. Dokumentation auf der Diskette im ASCII-Format.

Spezial X: CRC-Softwaresicherung
Fehlerüberprüfung bei Datenübertragung durch "Cyclic-Redundancy-Check" als reine Softwarelösung. Die gebrauchsfertigen Module lassen sich sowohl in Turbo Pascal, Assembler und Quick Basic 4.x einbinden. Eine Portierung für weitere Sprachen ist möglich, da alle Module im Quellcode vorliegen. Mit Dokumentation auf der Diskette.

Spezial XI: MAP, MARK & RELEASE
Die drei beliebten Utilities aus PASCAL zum Handling speicherresidenter PC-Programme in verbesserter Turbo-Pascal-4/5-Version! MAP zeigt die RAM-Belegung, verwendete Interrupts und freien Speicherplatz. MARK markiert, RELEASE löscht speicherresidente Programme. Die Diskette enthält den Quellcode in Turbo Pascal 4/5, die lauffähigen Programme und ausführliche Dokumentation.

Alle TOOLBOX-Spezial jeweils **nur 35,– DM***

Jetzt Demodiskette über alle "Spezials" erhältlich!

* **Unabhängig von der Anzahl der bestellten Artikel berechnen wir für das Inland 4,– DM bzw. für das Ausland 6,– DM Porto und Verpackung. Unverbindliche Preisempfehlung.**

Bitte benutzen Sie die Bestellkarte.

DMV-Verlag · Postfach 250 · 3440 Eschwege

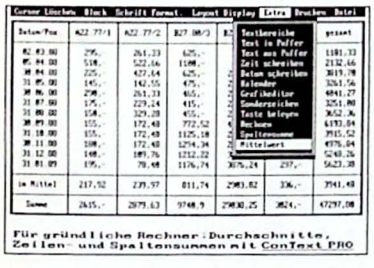
ConText PRO

Textverarbeitung mit Konzept

Das praxiserprobte Textprogramm für den professionellen Einsatz

Kompromißlose WYSIWYG-Darstellung
Einfache und blitzschnelle Bedienung
Vielseitigkeit und hohes Tempo
Extrem handlich und flexibel

Bedarf an Tabellen, Statistiken oder Aufstellungen?
ConText PRO kann's: Summenfunktionen und Durchschnittswerte zeilenweise und im Spaltenblock bearbeiten menügesteuert Ihre Zahlenwerte.



Zusammen mit dem eingebauten Grafikeditor für PC-Blockgrafik und in Verbindung mit Breit-, Fett-, Kursivschrift und Unterstreichen lassen sich mit ConText PRO aussagekräftige und übersichtliche Tabellen und Statistiken blitzschnell erstellen.

ConText PRO, das Rechengenie! Neben Summen und Durchschnitten steht Ihnen noch jede Menge Mathematik zur Verfügung: unter anderem Multiplikation, Division, Potenzieren, logische Operationen wie AND, OR und XOR sowie mathematische Funktionen wie Sinus, Logarithmus und Wurzelziehen.

MAUSTREIBER

Für IBM PC/XT/AT und Kompatible ab 256 KByte RAM und DOS 2.0. Spezialtreiber für CGA-, EGA-, VGA- und Hercules-Grafikkarten. Unterstützung aller Zwei-Tasten-Mäuse. 200 Seiten deutsches Handbuch im Ringordner.

* Unabhängig von der Anzahl der bestellten Programme berechnen wir für das Inland DM 4,– bzw. für das Ausland DM 6,– Porto und Verpackung.

Unverbindliche Preisempfehlung

Bitte benutzen Sie die Bestellkarte.

ConText-ADREWA

Universelle Adreßverwaltung

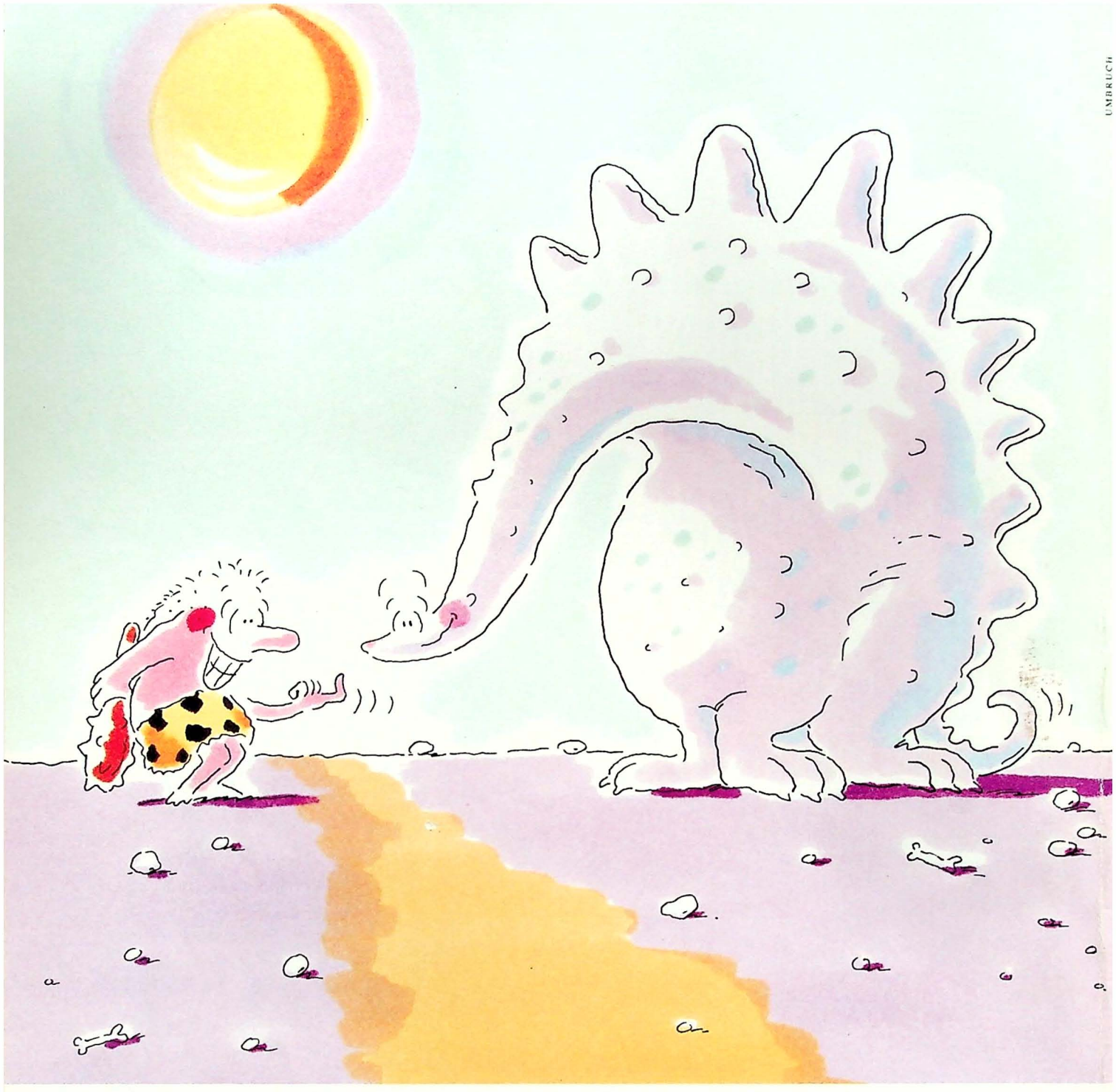
- Beliebiges Suchen nach einzelnen Adreßfeldern
- Markierungen mit Wildcards in allen Feldern
- Datenexport gemäß einer Exportliste wählbar
- Möglicher Datenimport aus ASCII-Dateien
- Kontextbezogene Hilfe bei allen Funktionen
- Adreßliste mit Online-Cursorsteuerung
- Bis zu dreizehn verschiedene Indexfelder
- Aktuelle Adresse in ConText PRO exportierbar

ConText-ADREWA zum Ausbau von ConText PRO und anderer Programme mit einer leistungsfähigen Adreßverwaltung. Durch eine spezielle Option können exportierte Datensätze direkt mit der ConText-PRO-Serienbrief-Funktion weiterverarbeitet werden. Für den einmaligen Gebrauch wird die aktuelle Adresse per Knopfdruck in ConText PRO übernommen. ConText-ADREWA kann wahlweise speicherresident oder als normales Programm installiert werden.

ConText PRO Das professionelle Textprogramm	199,– DM*
ConText PC Erprobte Version für den Einstieg	99,– DM*
ConText-ADREWA Universelle Adreßverwaltung	49,– DM*
Update auf ConText PRO	129,– DM*
Demonstrationsdiskette	5,– DM

DMV-Verlag · Postfach 250 · 3440 Eschwege

DMV Daten- und Medienverlag



David. Die pfiffige Datenbank.

David bewältigt auch große Probleme auf Anhieb. Denn David 1.0 ist das neuartige dBASE-III-Plus-kompatible Datenbanksystem mit der bedienerfreundlichen SAA-Oberfläche und umfangreichen starken Ex- und Importmöglichkeiten, z. B. in Lotus-Dateien und SDF.

David bedeutet: freie Struktur-Definitionen von Datensätzen; Suchfragen auf drei Ebenen zugleich; Sortieren mit und ohne Index in beliebig vielen Ebenen; viele Funktionen zu Berechnung, Kalkulation und relationaler Verknüpfung; Befehlsmakros.

David zeigt Ihnen alle Daten tabellarisch oder in frei definierbaren Eingabemasken. Arbeitssituationen speichert er als wiederholbare Einstellungen. Was Sie wollen, gibt er zum Druck: in Tabellen, in Masken oder als Report mit vielfältigen Optionen.

David, die schlagkräftige Datenbank, erhalten Sie für 498 Mark*.

Coupon senden an: STAR DIVISION, Zum Elfenbruch 5–11, 2120 Lüneburg.

STARDIVISION

INFO-COUPON

Bitte senden Sie mir nähere Informationen über David und das umfangreiche Software-Angebot von STAR DIVISION.

Name, Vorname

Straße, Platz

PLZ, Ort

* Unverbindliche Preisempfehlung Erwähnte Warenzeichen: dBASE (Ashton Tate), Lotus (Lotus Inc.).